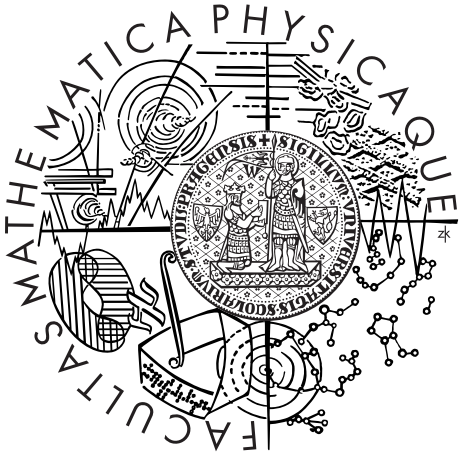
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Marek Vašut

Green Clusters

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Leo Galamboš, Ph.D.
Study programme: Computer Science
Specialization: Software Systems

Prague 2014

I would like to thank my supervisor, RNDr. Leo Galamboš, Ph.D., for his advice, guidence and for keeping me motivated. I would like to thank namely, Prof. RNDr. Jaromír Antoch, CSc. for numerous advices on statistical processing of measurements, RNDr. František Lustig, CSc. for ideas on improving the measurement equipment and method accuracy, Detlev Zundel for proof-reading the text and both him and Andreas Widder for being supportive friends and great bosses. Last but not least, I would like to thank my parents for all the help and support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Green Clusters

Autor: Marek Vašut

Katedra: Department of Distributed and Dependable Systems

Vedoucí diplomové práce: RNDr. Leo Galamboš, Ph.D.

Abstrakt: Práce se zabývá možnostmi snížení spotřeby soudobého počítačového clusteru pomocí použití hardwarových komponentů s nízkou spotřebou.

Na clusteru je provozována implementace algorithmu Map-Reduce a výpočetní jednotky jsou sestaveny ze systémů s procesory ARM nebo systémů s procesory kombinujícími ARM a FPGA v jednom pouzdře. Chování clusteru je rozebráno z pohledu jak výpočetního výkonu, tak z pohledu spotřeby energie.

Práce se zabývá problémy a obtížemi objevujícími se při integraci systémů na bázi architektury ARM, hlavně pak kombinovaných systémů obsahujících k ARM ještě FPGA, do frameworku Map-Reduce. Samotný Map-Reduce framework je podroben zkoumání, aby byly identifikovány nejhrubší problémy při jeho nasazení v prostředí architektury ARM.

Klíčová slova: Map-Reduce, Energy-efficient, ARM, FPGA, Big-Data

Title: Green Clusters

Author: Marek Vašut

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Leo Galamboš, Ph.D.

Abstract: The thesis evaluates the viability of reducing power consumption of a contemporary computer cluster by using more power-efficient hardware components.

The cluster in question runs an Map-Reduce algorithm implementation and the worker nodes consist of either systems with an ARM CPU or systems which combine both an ARM CPU and an FPGA in a single package. The behavior of such cluster is discussed from both performance side as well as power consumption side.

The text discusses the problems and peculiarities with the integration of an ARM-based and especially the combined ARM-FPGA-based systems into the Map-Reduce framework. The Map-Reduce framework performance itself is evaluated to identify the gravest performance bottlenecks when using the framework in the environment with ARM systems.

Keywords: Map-Reduce, Energy-efficient, ARM, FPGA, Big-Data

Contents

Introduction

The contemporary data centers which process large sets of data require extreme amounts of electric energy. Most of this energy is turned into waste heat by the energy-inefficient systems used within these data centers.

A lot of research went into optimizing the power consumption and energy efficiency of modern data centers, both on the side of using more energy-efficient systems and on reusing the waste heat for other purposes, like heating of buildings. The author believes this field still contains further opportunities for improvement.

This text researches the reduction of power consumption of modern data centers by using components which are explicitly designed for both high performance and energy efficiency, and leveraging their specific features to accelerate the processing of data. This different approach to energy saving has seen only little coverage so far.

It is yet unclear whether it is viable to use a contemporary ARM system or cluster of such to replace a power-demanding x86 system. It is also unclear whether the cluster of such ARM systems can deliver the same processing power as the x86 system while consuming less energy.

The scope of such topic is extremely broad, so instead of trying to answer the general question, this text will focus on answering the question for a particular setup, which approximates a typical use case.

The research conducted in this text focuses in particular on the viability of replacing a server-grade x86 system in a cluster with one or multiple ARM systems and the impact of doing so on the power consumption of such a cluster.

The particular application running on the cluster is the Map-Reduce programming paradigm, which is well understood and well analyzed already. Map-Reduce is also deployed in real-world cluster workload scenarios, therefore using this particular paradigm provides for a fine approximation of the usual use case.

Since the ARM systems themselves are currently much more fragmented than the regular server-grade x86 systems, this text does discuss the peculiarities of using an ARM system in such a cluster setting. It also discusses the options of using features specific to these ARM systems, which are available to accelerate the speed of data processing and the impact of doing so on the power consumption.

Motivation

Initially, the motivation to investigate this option came from constant woes of my relatives, who complain that my computers consume too much energy. By looking at the bigger picture, it becomes clear that the power consumption of these few computers is absolutely negligible at best.

There are places, where there are lots of computers, which do repetitive tasks. These are the data centers, which run parallel computational jobs. Saving even a small amount of energy on each of those computers in such data centers does, due to the sheer numbers of those computers, have a potential to create a difference.

The author's background is not related to Big-Data. Instead, the author has worked in the field of operating systems and has been living on the border between

the hardware and the software. The author hopes that this shall provide for an interesting and innovative take on the topic of data center power consumption.

Both Altera and Xilinx recently introduced a chip which combines both an ARM CPU core and a dedicated FPGA part. It became clear that this might just be the right chip to reduce the power consumption of a data center by moving parts of the computation into specialized hardware.

The goals of the thesis

The goals of the thesis are:

1. Integrate ARM and ARM-FPGA system into Map-Reduce framework
2. Evaluate x86, ARM and ARM-FPGA in the Map-Reduce context
3. Identify problems with ARM and ARM-FPGA in Map-Reduce context

Structure of the text

The first two chapters are purely introductory. **Chapter 1** presents a detailed breakdown of the task and discusses previous work related to this text. **Chapter 2** starts the theoretical part of the text with a brief explanation of the Map-Reduce algorithm principles.

The next couple of chapters discuss the tools and methods used for analyzing the problem at hand. **Chapter 3** explains the selection of hardware for the reference cluster used in this text. **Chapter 4** lists available Map-Reduce frameworks, discusses their pros and cons and details the framework selected for testing conducted in the rest of the text. **Chapter 5** presents a detailed explanation of a Map-Reduce benchmark used to evaluate the distinct systems used in this text. Finally, **Chapter 6** concludes the theoretical part by discussing the methods used for quantifying the power efficiency.

The following two chapter discuss necessary implementation details. **Chapter 7** contains detailed explanation of the specifics of using the FPGA to accelerate computation in general. **Chapter 8** then explains the implementation details of the selected Map-Reduce benchmark for both software and hardware-assisted case based on the data from previous chapters.

Finally, with all the infrastructure in place, the remaining chapters evaluate the behavior of the systems from both timing and power efficiency angles. **Chapter 9** contains evaluation of the benchmark results on a single node configurations to identify the gravest issues on all the platforms and drive further investigation. Based on the results, **Chapter 10** presents the evaluation of the benchmark results on a single node for different parameters, which are of interest, **Chapter 11** discusses the evaluation of the benchmark on a full ARM cluster and **Chapter 12** dissects the performance considerations of a combined ARM-FPGA system.

1. The problem in detail

As was already outlined in the preface, the power consumption of modern data centers is enormous. The capability of contemporary CPUs to cater for growing demand for processing power is limited. It is therefore necessary to start searching for alternative approaches to handling the demand.

1.1 The question

The big question this text tries to answer is:

1. *Is it viable, at least for certain workloads, to replace a server-grade x86 system with a power-efficient ARM system or cluster of those?*

The question is of a very general nature. It is therefore important to limit the scope of the research to a smaller part of it and analyze this subset thoroughly. The rest of this chapter will precisely define which particular parameters of the above question will be fixed and thus define the the perimeter of the research carried out by the rest of the text.

Firstly, an important parameter is the selection of hardware. This text uses three particular test systems. One is a reference x86 system to represent the typical server-grade Big Iron hardware and two are power-efficient contemporary ARM systems. The particular x86 system is an Intel® i3970X system, while the two ARM systems are a Freescale® i.MX6 Quad and Altera® SoCFPGA Cyclone V.

The selected x86 system ought to give a reasonable approximation of consumption and performance characteristics of a typical data center grade computational node, while the two ARM systems represent the power-efficient components aiming to replace the x86 systems.

The Freescale i.MX6 Quad system is used to research whether an ARM core itself or a cluster of such can be used to outperform the x86 system under certain conditions. The Altera SoCFPGA Cyclone V system is used to research if it is possible to use ARM with an FPGA to outperform the x86 system in certain tasks.

2. *The evaluated hardware consists of a reference x86 system, an ARM system and a combined ARM-FPGA system*

With the hardware selection fixed, there are still many parameters that can be varied. This text uses the Map-Reduce programming paradigm as the cluster workload. While there are other means to process large sets of data on a cluster available today, this text focuses solely on Map-Reduce.

There are two good reasons for using Map-Reduce. Firstly, Map-Reduce is an integral part of real-world cluster workloads today. A good example is the Apache™ Hadoop® PoweredBy article [22], quoting from the article:

> Facebook: We use Apache Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning.

> Twitter: We use Apache Hadoop to store and process tweets, log files, and many other types of data generated across Twitter. We store all data as compressed LZO files.

Secondly, Map-Reduce is easy to implement and easy to deconstruct into separate steps. This in turn allows very fine grained measurements of every processing step.

3. *The programming paradigm used throughout the text is Map-Reduce.*

Due to the vast amount of options onto which the Map-Reduce paradigm can be applied, it is important to select a reasonable benchmark job for the Map-Reduce cluster.

The benchmark used throughout this text is the inverse MD5 function, i.e. to search for a plain text, that generates a supplied MD5 hash. Since the MD5 hashing function is, just like any other hashing function, designed to be unidirectional. There is thus no (known) algorithmic way of doing this. Excluding cryptographical attacks one option to determine the input text for a given MD5 hash is to exhaustively search the complete input space. This is done by computing MD5 hashes for all possible inputs and test for equality of the computed and the given MD5 hash.

Conducting a computation of MD5 hashes for all possible inputs is not viable, but it is possible to compute all MD5 hashes if the set of inputs is limited. This text thus uses a bounded set of inputs for the MD5 benchmark.

There are multiple reasons for selecting such a benchmark. A detailed explanation follows in Chapter 5 on page 20, yet the main reason for selecting this particular benchmark is the following. The MD5 computation is a CPU-bound benchmark and the computation itself is very well understood. The search for the matching MD5 hash is in turn an I/O bound benchmark. Furthermore, the data on which the benchmark operates can be divided easily and evenly in between cluster nodes.

4. *The benchmark is a search for a plain text, that generates a specified MD5 hash. The search is conducted for a bounded set of input data.*

1.2 Related work

There are papers dealing with optimizing the power consumption of a cluster by using ARM systems instead of conventional x86 systems.

An interesting paper is *Tibidabo: Making the case for an ARM-based HPC system* [7] by N. Rajovic at al., where the authors simulated a theoretical cluster of ARM systems and then also physically built one from a limited amount of ARM systems. The paper presents performance figures and comparisons to a mobile Intel Core chip, deeming the approach of building an ARM cluster viable and worth researching further. The cluster in this case was built from nVidia Tegra2 systems and uses 100 Mbit/s USB ethernet to communicate in the cluster, both of which are sub-par and outdated solutions, yet this in itself adds to the viability of using ARM systems in a clustering solution.

A similar clustering attempt was done on a smaller scale, but with a more up-to-date hardware in a paper by Z. Krpić et al. titled *Towards an energy efficient*

SoC computing cluster [5]. The comparison was done between Intel Pentium 4 and an Allwinner A20 system, with the latter again using 100 Mbit/s USB ethernet. The paper points out that the cluster suffered network bottlenecks under heavy load, yet the network bottleneck can be solved by a better choice of hardware. There is also a very nice description of the power consumption measurement equipment contained in this paper. It again confirms that using an ARM system in a cluster setting is viable.

Finally, a paper by Z. Ou et al, titled *Energy- and Cost-Efficiency Analysis of ARM-Based Clusters* [6] evaluated a cluster of systems based on the Texas Instruments OMAP4 CPU for distinct CPU-bound and memory-bound workloads and observed an energy efficiency in comparison to an Intel system. The paper points out that the energy efficiency of the ARM systems varies substantially depending on the particular workload.

All of the papers above focused on ARM clusters in general, yet neither of them researched the usability of ARM systems in the context of the Map-Reduce algorithm. There have been attempts to apply Apache™ Hadoop® onto ARM systems by Heng Yan in *The first Spark/Hadoop ARM cluster runs atop Cubieboards* [10] and Yanjun Wang in *Hadoop MapReduce Performance Study on ARM cluster* [9]. The results from both experiments conclude that the overhead of the Apache™ Hadoop® poses a serious impediment for the ARM systems used in the tests.

There are also very interesting papers dealing with accelerating the Map-Reduce computation by using dedicated hardware. The most prominent target of such offloading seems to be a GPU.

There is a paper about using Apache™ Hadoop® in combination with OpenCL from M. Grossman et al. titled *HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop® and OpenCL* [3], which deals with accelerating the Hadoop® computation on an OpenCL-capable hardware.

There is similar research by Chen He and Peng Du titled *CUDA Performance Study on Hadoop MapReduce Clusters* [4], which uses nVidia CUDA instead and which even points out that using GPUs to accelerate the Map-Reduce computation has a positive effect on the overall power consumption.

Both of the approaches above can only be applied to a well defined subset of the problems which can be solved by the Map-Reduce algorithm, yet both prove that a hardware accelerated Map-Reduce is viable. The OpenCL paper is even more interesting, since Altera recently started promoting an OpenCL SDK, that allows running OpenCL programs on the FPGA. Unfortunately, since using the OpenCL SDK requires special license, it will not be used in the rest of the text.

Finally, there is already a small number of papers dealing with accelerating the Map-Reduce algorithm using FPGAs. A paper by Dong Yin titled *Scalable MapReduce Framework on FPGA Accelerated Commodity Hardware* [11] is using a NetFPGA FPGA box coupled with a control computer to perform the computation. The control computer runs virtual Map-Reduce tasks, that are internally sent over the network to the NetFPGA boxes to do the processing. The downside with this approach is that the FPGAs must deal with the network communication. Also the paper does not deal with power consumption.

Another interesting paper is from *Zhongduo Lin* titled *ZCluster: A Zynq-based*

Hadoop cluster. This work is extremely important for this text, since it is dealing with accelerating Apache™ Hadoop® on a combination of ARM and FPGA. The paper points out that there is a huge performance benefit of using the FPGA, yet the paper does not deal with power consumption of the resulting cluster. Moreover, the paper explicitly states that the FPGA must be pre-programmed with a configuration upon boot of the system, which makes the result inflexible.

Even with this previous work there are still a lot of open research questions connected to using a Map-Reduce implementation on a system combining an ARM CPU and an FPGA and the resulting power consumption.

2. The Map-Reduce algorithm

The original Map-Reduce algorithm was proposed by Google's Jeffrey Dean and Sanjay Ghemawat in paper *MapReduce: Simplified Data Processing on Large Clusters* [1] in December 2004. The Map-Reduce algorithm is designed to allow parallel processing of big amounts of input data on a large number of interconnected computers, each of which can fail at any time.

The purpose of a Map-Reduce implementation is to provide the programmer with a unified interface, which shields the programmer from handling the parallelization of tasks, distribution of the processing tasks across the cluster, managing the load balancing in the cluster and handling the redundancy in case a node in the cluster fails. A Map-Reduce implementation let's the programmer focus solely on implementing the data processing part of the task.

To explain Map-Reduce properly, it is necessary to start from the basic building blocks. A typical Map-Reduce cluster consists of four main components:

1. Nodes
   The Computer nodes execute tasks which process sets of data. These *Node*s are connected together via a network to form a cluster. Each *Node* can fail at any time and the Map-Reduce implementation has to handle such a case. A processing task running on a *Node* is called a *Worker*.

2. Master node
   The *Master* node is responsible for assigning tasks to the *Worker*s, for tracking completion status of the tasks assigned to the *Worker*s in the cluster and it acts as an arbiter for the whole cluster. In case a *Node* in the cluster fails, the *Master* is responsible for rescheduling tasks from *Worker*s on that *Node* to other *Worker*s. The *Master* node can fail at any time as well and a Map-Reduce implementation often provides redundancy options.

3. Global storage
   The purpose of *Global storage* is to provide sets of input data to *Worker*s and to store final results of the Map-Reduce computation. The *Global storage* usually does not contain intermediate results of the Map-Reduce transformation.

4. Network interconnect
   The *Network* interconnect between all the *Node*s in the cluster assures that all of the nodes can access one another and also the *Global storage*. The *Network* is usually[1] a flat network segment where all nodes on the segment are equal. It is possible that a *Network* failure may occur and the Map-Reduce implementation has to handle such case.

Any Map-Reduce computation, taking place on the cluster outlined above, consists of three distinct stages. First is the *Map* stage, then the *Intermediate* stage and finally the *Reduce* stage. During the *Map* and *Reduce* stages, *Map* and *Reduce* tasks are assigned to the respective *Worker*s. The *Intermediate* stage is

[1] Scaling beyond the size of single network segment is often done using a multi-layer Map-Reduce job.

special in that it is running on each *Node* and is responsible for post-processing the intermediate local data resulting from the *Map* stage.

The implementer is responsible for providing the implementation of the *Map* and *Reduce* tasks. The implementer is also responsible for making the input data for the Map-Reduce computation available to the *Worker*s via the *Global storage*. Finally, the implementer needs to provide a space on the *Global storage*, where Map-Reduce can store the results of the computation.

Finally, note that a particular *Node* in the cluster can host multiple *Worker*s. The tasks assigned to a *Worker* can be executed either in parallel or in sequence. The order in which *Worker*s execute the tasks is completely irrelevant as the tasks must have no interdependence.

2.1 The Input data

The *Input data* consists of one or more files, which are stored on the *Global storage*. Internally, these files are organized as a list of (key, value) pairs. These *Input data* are all read-only and are never written to. The Map-Reduce implementation represents the *Input data* as a concatenation of all the (key, value) pairs in all of the input files in an undefined order.

It is expected that the *Input data* are large, often so large that they cannot be loaded into the RAM of any single *Node* and often even into the local storage of a single *Node*.

A subset of these *Input data* is called a *Chunk*. Such *Chunk* is passed onto a *Worker* running a *Map* task for processing. The *Chunk* should be small enough so that it fits into the RAM of the target *Node* to eliminate overhead spent on doing block I/O.

It is up to the program which splits the *Input data* into the *Chunk*s to make sure that each (key, value) pair is placed into exactly one *Chunk*.

The *Master* node assigns the *Chunk*s to *Worker*s running a *Map* task. The *Master* node therefore has an exact knowledge of the mapping between the *Chunk*s, the *Worker*s and the *Node*s. The *Master* can thus reschedule a processing of a *Chunk* to another *Worker* in case a *Node* fails.

2.2 The Map stage

Throughout the *Map* stage, the *Master* node assigns the *Map* tasks to the *Worker*s. To make naming shorter, a *Worker* performing a *Map* task will henceforth be called a *Mapper*.

A data flow on a single *Node* during the *Map* stage is presented at the upper half of Figure 2.1. The *Intermediate* stage and *Shard*s are explained shortly in the next Section 2.3. The splitting of the *Input data* into *Chunk*s is not considered part of the *Map* stage, it is part of the graphic representation for the sake of completeness of the idea.

Each *Mapper* has a *Chunk* of *Input data* associated with it. The *Mapper* iterates over the (key, value) pairs in the *Chunk* and performs the *Map()* function on each of the pairs.

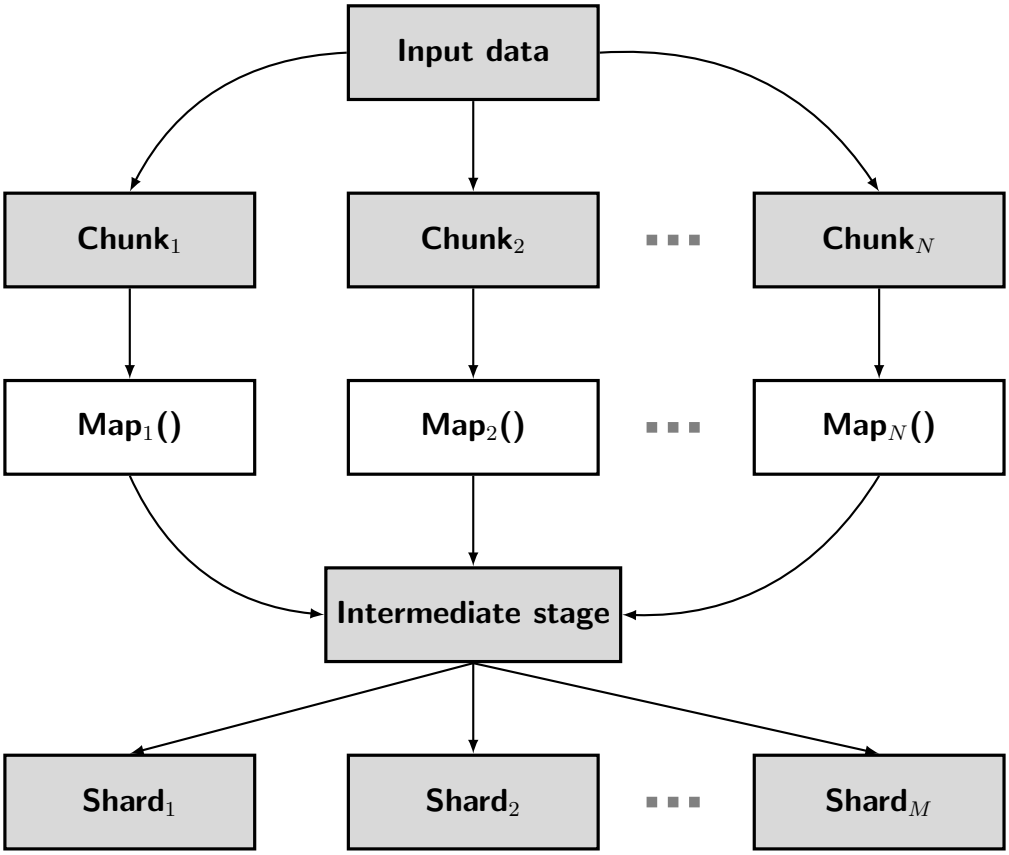


Figure 2.1: Map and Intermediate stage pipeline on a single Worker

The *Map(key, value)* function takes a (key, value) pair as an input parameter and produces zero or more intermediate (key, value) pairs. These intermediate (key, value) pairs are passed into the *Intermediate stage*.

2.3 The Intermediate stage

The *Intermediate* stage consists of multiple operations, most important of which are the *Combining* and *Shuffling*.

While there can be multiple *Mapper*s running on a *Node* in parallel, the operations in the *Intermediate* stage usually operate on the results produced by all of the *Mapper*s running on that *Node*, which perform part of the same Map-Reduce Job. The *Intermediate* stage can thus start only once the *Mapper*s generated enough (key, value) pairs.

The first of the operations is *Combining*. The task of the *Combining* is to iterate over all of the intermediate (key, value) pairs and aggregate all of the (key, value) pairs sharing the same key. The result from the *Combining* is a list of (key, list of values) pairs. The effect of *Combining* is beneficial because all of the entries, which share the same intermediate key, usually end up being processed by the same *Reducer*.

The *Combine()* function can be replaced by the implementer in case a finer control over the Map-Reduce algorithm is needed. The *Combine()* function has an iterator over the intermediate (key, value) pairs and produces a list of (key, list of values) entries as a result.

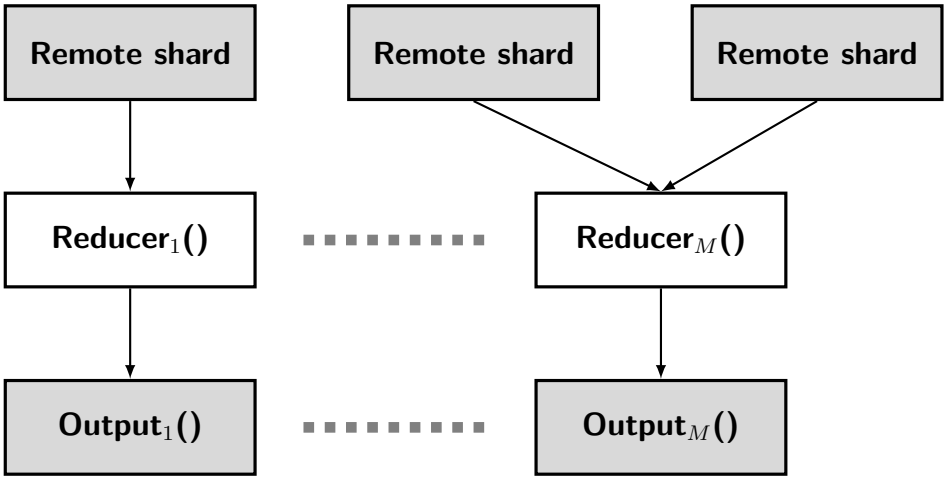


Figure 2.2: Reduce stage pipeline on a single Worker

The next operation happening after *Combining* is called *Shuffling*. This operation is responsible for dividing the whole bulk of intermediate local data resulting from *Combining* between the *Reducer*s. The *Shuffling* is responsible for dividing the data evenly, so that no Reducer is unnecessary overloaded with data. The *Shuffling* is also often responsible for dividing the data such that neighboring or related keys are delegated to the same Reducer.

The data sent to the *Reducer*s are called *Shard*s. The format of the *Shard* is such that it contains one or a few keys and each key has multiple values associated with it. The *Shard* is essentially a fraction of output from the *Combiner*.

2.4 The Reduce stage

Finally, throughout the *Reduce* stage, the *Master* node assigns the *Reduce* tasks to the *Worker*s. To make naming shorter, a *Worker* performing a *Reduce* task will henceforth be called a *Reducer*. A data flow on a single *Node* during the *Reduce* stage is presented in Figure 2.2.

The *Reducer* operates on one or multiple *Shard*s of data. The *Reducer* executes the *Reduce()* function for each key in all the *Shard*s available to the Reduce task. The *Reduce(key, iter)* function has two parameters, the key already mentioned and an iterator that is used by the *Reduce()* function to iterate over all values associated with the key.

It is important that there exists a *Reducer* for every key present in each *Shard*. Every entry in each *Shard* must be processed by some *Reducer* so that no data are lost.

The output of the *Reducer* is often a very small amount of (key, value) pairs. These (key, value) pairs are stored on the *Global storage* as the final result of the whole Map-Reduce computation.

Note that it is well possible that the size of the set of values associated with a particular key can be bigger than the amount of RAM of the *Reducer*. In such case, the *Reducer* has to use it's own local storage and use external memory operations to work with such large set of data.

2.5 The Output data

The results of all of the *Reducer*s are stored on the *Global storage*, from where they can be retrieved by the implementer of the Map-Reduce job or the user. The size of the *Output data* is often very small compared to the size of the *Input data*.

Every *Reducer* produces a separate set of output (key, value) pairs on the *Global storage*. It might therefore be needed to do a post-processing pass on the resulting separate output (key, value) pairs. The post-processing pass is out of the scope of the Map-Reduce algorithm and is up to the implementer to implement.

3. Cluster configuration

3.1 Software setup

This text uses the GNU/Linux operating system for all practical measurements. This selection is based on two very important facts.

Firstly, GNU/Linux is a native environment for the Map-Reduce algorithm. Both the original Google Map-Reduce design proposed by Jeffrey Dean and Sanjay Ghemawat in paper *MapReduce: Simplified Data Processing on Large Clusters.* [1], the Apache™ Hadoop® and other Map-Reduce implementations listed in Chapter 4 on page 17 are native GNU/Linux programs.

Secondly, GNU/Linux is the most accessible platform running on power-efficient architectures, which can host a Map-Reduce node. While there are other operating systems available on these architectures, adapting the Map-Reduce framework to such operating systems would bring new unknowns into the already complex problem.

3.2 Hardware setup

The hardware used as test beds in this text consists of three systems. These particular systems were selected because of their largely distinct properties, but also because it is easy to conduct measurements using these systems. Below is an overview of the selected platforms and further sections describe them in detail.

1. Freescale® i.MX6 Quad , armv7a
2. Altera® SoC-FPGA , armv7a with FPGA
3. Intel® Core™ i7, x86-64

3.2.1 ARMv7a: Freescale® i.MX6 Quad

The particular board with this CPU is the Kosagi Novena. The board has the following configuration:

- CPU: Freescale® i.MX6 Quad @ 1200 MHz
- L1 cache: 32kiB I-Cache / 32kiB D-Cache per physical core
- L2 cache: 1 MiB L2 cache shared by all cores
- Memory: 4 GiB DDR3 @ 1066
- Storage: 64 GiB SDXC card
- Network: Freescale® FEC

This system is an ARMv7a architecture. This particular system was picked because it was the only obtainable board featuring 4 GiB of DDR3 DRAM and an ARMv7a CPU at the time of writing.

This commercially available Freescale® i.MX6 Quad chip contains four ARM Cortex-A9 CPU cores, rated at up to 792 MHz core clock speed. The Novena board is populated with a CPU from a limited series rated at up to 1200 MHz core clock speed. It shall be noted that all four CPU cores are connected in the same CPU cluster and share the same high-speed interconnect to the DDR3 memory and the peripherals.

The memory subsystem of this chip is also noteworthy. The Level 2 cache, shared between the cores, is big compared to other ARMv7a CPUs. Unfortunately, there is a drawback as well. The i.MX6 Quad can only clock the DDR3 DRAM module at up-to 532 MHz, being equal to a 1066 DDR3 speed rating. Modern DDR3 modules can run at up to 1600 speed rating. Details on the DDR3 memory behavior can be found in a paper from Ulrich Drepper titled *What Every Programmer Should Know About Memory* [2].

The i.MX6 Quad also contains an integrated gigabit ethernet controller, the Freescale® FEC. The FEC is not a server-grade ethernet controller. It is unknown what the performance implications of using the FEC ethernet controller are.

3.2.2 ARMv7a: Altera® Cyclone V SoC

The particular system used here is the combination of a DENX MCV SoM[1] inserted into the DENX MCVEVK baseboard[2]. Unfortunately, the board used is still a prototype and it was not possible to obtain more boards.

- CPU: Altera® Cyclone V SoC C6 @ 800 MHz
- L1 cache: 32kiB I-Cache / 32kiB D-Cache per physical core
- L2 cache: 512 kiB L2 cache shared by all cores
- Memory: 1 GiB DDR3 @ 667
- Storage: 4 GiB eMMC and external USB pen drive
- Network: DesignWare DWMAC

This system is also an ARMv7a architecture. This particular system was picked because this is one of the few easily available boards featuring the Altera® Cyclone V SoC processor including an FPGA.

The Altera® Cyclone V SoC contains two ARM Cortex-A9 CPU cores rated at up to 800 MHz core clock speed. Both cores are in the same CPU cluster and share the high-speed interconnect to the DDR3 memory and peripherals.

The interesting feature of this particular CPU is the integrated FPGA. The Cyclone V FPGA in this chip is connected into the same high-speed interconnect

[1] System-on-Module, the CPU, RAM and storage board

[2] Breakout board with necessary connectors (Ethernet, serial console, power supply, ...)

as is the CPU and RAM. This configuration of the in-CPU interconnects promises a potential for a very good performance.

It is clear that this system does not have a sufficient amount of storage to host any Map-Reduce task which generates much data. This particular problem is solved by attaching an external USB pen drive.

3.2.3 x86-64: Intel® Core™ i7

This system is selected as a reference machine. The data centers of today often use Intel® Core™ CPUs or Intel® Xeon™ CPUs to run Apache™ Hadoop® or similar heavy-weight implementation of the Map-Reduce. To be able to compare the measurements of the ARM systems conducted in later chapters, an reference system is used.

This particular system has the following configuration:

- CPU: Intel® Core™ i7-3970X CPU @ 3.50GHz
- L1 cache: 32kiB I-Cache / 32kiB D-Cache per physical core
- L2 cache: 256 kiB L2 cache per physical core
- L3 cache: 15MiB L3 LLC shared by all cores
- Memory: 64 GiB DDR3 , Quad-channel configuration
- Storage: 128 GiB Samsung® 840 EVO SSD
- Network: Intel® Corporation 82579LM

This system is, unlike the rest, an x86-64 system. As was mentioned previously, it will be used as a reference for the results measured on the previous two power-efficient ARM systems and put them in proportion to the computational power of a contemporary data center node.

The system has a powerful CPU with 6 logical cores and 2 threads of execution per core. This text will not discuss the impact of the HyperThreading on the computational performance of the worker node.

The system also has a server-grade network interface, which would be useful when comparing the network performance of the FEC and DWMAC. The server-grade high-performance SSD storage will in turn be useful in comparison to the performance of local storage of the power-efficient nodes.

4. Map-Reduce implementations

4.1 Apache™ Hadoop®

The Apache™ Hadoop® [13] is considered to be the de-facto reference implementation of the Map-Reduce algorithm. It is not an implementation of Map-Reduce algorithm in itself, but a framework consisting of four core components – Hadoop® Common, HDFS, YARN and Hadoop® Map-Reduce.

It is the Hadoop® Map-Reduce which implements the Map-Reduce algorithm itself. The HDFS provides a fail-safe distributed file system, YARN provides resource management and the Hadoop® Common is a set of libraries shared by all of those components. The whole Apache™ Hadoop® platform is an even bigger set of projects, but it is far beyond the scope of this text to discuss those.

There are multiple reasons why Hadoop® was not selected for this text. Most importantly Hadoop® itself is big in all aspects. The overhead of Hadoop® is not a problem on a big x86 system, which has an abundance of performance, but this is far from being the case on a small ARM system. This was already pointed out in works by Heng Yan titled *The first Spark/Hadoop ARM cluster runs atop Cubieboards* [10] and Yanjun Wang titled *Hadoop MapReduce Performance Study on ARM cluster* [9].

On a more controversial note, Hadoop® is implemented in Java. Since the author is not a Java expert and therefore cannot provide relevant performance evaluation of the implementation in Java, this became one of the major issues not to use it for this work.

Finally, some of the ARM systems used in this text use external accelerators, like an FPGA, to speed up the computation of the Map-Reduce job. These accelerators tend to have various obscure requirements, requiring the control code to be written in a language offering a more fine grained control of the execution, like for example C. While it is possible to call C functions or execute external helpers from Hadoop® , this option is rather complicated and was not pursued.

4.2 Twister

Similar to Hadoop® , Twister [24] is also written in Java, but compared to Hadoop® , Twister claims to be much more lightweight. Twister is not explicitly a generic Map-Reduce framework though.

The Twister framework is called an Iterative Map-Reduce framework. This means that Twister is optimized for running the same Map-Reduce job in a loop many times. The basic premise of Twister is that the data used in it's Map-Reduce computation can be divided into two groups – one being Static data and the other being Dynamic data.

The Static data are pushed into the Map-Reduce cluster once and these data stay on the respective Worker nodes. The Dynamic data are updated in the cluster during every iteration. It is expected that the Static data represent the vast majority of the data used during the single Map-Reduce iteration and therefore the amount of data updated during each iteration is small.

Since this text researches a generic Map-Reduce implementation and since the author is not familiar with Java, this framework will not be used in this text.

4.3 BashReduce

An interesting project, which mitigates the problem with controlling hardware from within the Map-Reduce implementation is BashReduce [14]. Compared to Apache™ Hadoop® , BashReduce is only and solely an implementation of the Map-Reduce algorithm.

BashReduce is implemented in a BASH scripting language and uses BASH shell and related utilities to do the processing. Some of the utilities are not fully portable across UNIX systems, but since GNU/Linux was selected as the base operating system, this property is not a serious problem.

BashReduce also allows to easily control the underlying hardware, since it can easily execute an external tool written in C to interact with the hardware. Even elevating permissions to allow the external tool to interact with hardware is not much of a problem when using BashReduce.

On the other hand the fact that it is written in BASH is also the biggest problem with BashReduce. The BASH, just like any other shell scripting language, is an interpreted language and this does impose serious performance penalty. The author of BashReduce mentions this problem in the documentation [15] and tries to mitigate it by implementing some parts of BashReduce in C.

Furthermore, the problem with BashReduce is that it is completely missing any abstract means to package and transport additional files to remote nodes with the task itself. This makes it difficult to package for example the FPGA bitstream or an external program with the Map-Reduce job.

4.4 MARS

Closer to a hardware-assisted Map-Reduce implementation is a project called MARS [20]. The MARS project is implemented in C++, allowing for easy profiling and it is also very easy to implement extensions interacting directly with hardware.

The downside of the MARS project is the fact that it explicitly targets GPUs and even more explicitly, it focuses on implementing Map-Reduce using the nVidia CUDA technology. As of the time of the writing of this text, the nVidia CUDA on ARM is only available for the nVidia Tegra K1 systems, which are unfortunately not available to the author.

4.5 Disco

Disco [17] is an implementation of the Map-Reduce algorithm implemented initially by Nokia. This particular implementation is written mostly in Erlang and Python and has plenty of nice features, which make it interesting for this text.

Firstly, Disco already has an interface to implement both the Map() and the Reduce() functions by calling either an external program or by loading an external

shared library and calling it's functions directly. This feature is extremely helpful when interacting directly with hardware.

Furthermore, it is possible to package arbitrary files alongside a Disco Map-Reduce job and these files are automatically distributed to the target nodes. This again is very helpful when distributing for example configuration data for the FPGA for a given task.

One thing which is of great concern is the performance of Python and Erlang on ARM, which is yet unknown. It is well possible that the bottleneck of the entire Map-Reduce implementation will be the framework itself and not the computation.

On the other hand, the well known Python GIL [23], the mutex in CPython, which prevents multiple native threads from executing Python bytecode at once, does not pose a problem. This does sound surprising, especially since Disco runs on multi-core systems on multiple cores in parallel. Disco does not use threads on such multi-core systems though, instead, Disco spawns multiple instances of Python for each Worker, which thus completely avoids any adverse impact of the GIL.

This text uses Disco as the Map-Reduce implementation, since the initial evaluation of the available frameworks leaves Disco as the most promising option.

5. Map-Reduce benchmark

5.1 Benchmark detail

The benchmark used in this text is a search for the plain text that was used to generate an MD5 hash supplied by the user. The MD5 hash function is by definition a one-way function mapping plain text to a MD5 hash, ie. it is not possible to invert the MD5 function in an algorithmic way.

The benchmark thus implements a brute-force search. The MD5 hashes of all possible plain text input data are computed and compared to the supplied MD5 hash. Since the domain of all possible plain text inputs is infinite, computing MD5 hashes of all possible inputs would not be viable. The set of possible input data for the benchmark is thus explicitly bound. This test is rather synthetic, yet it also has very good properties when researching the viability of Map-Reduce on power-efficient hardware.

The input data for the benchmark is a list of all possible binary strings of a fixed length and a single MD5 hash. The output of the Map-Reduce benchmark is zero or more binary strings, namely those mapping to the supplied MD5 hash.

In case no output is returned by the benchmark, the supplied MD5 hash was generated from a binary string of a different length than the input strings. In case exactly one string is returned, the string maps to the supplied MD5 hash. Finally, the unlikely event of the transformation returning multiple distinct strings would mean that a hash collision was found and that there are multiple, different, strings mapping to the same MD5 hash. The returned strings are then exactly the colliding strings.

The benchmark tests two operations, both of which can be well parallelized, making them ideal for Map-Reduce. The operations are the application of the same algorithm over a large set of independent blocks of data and a search for an entry in a large set of independent data.

The computation of an MD5 hash over a large amount of independent blocks is a CPU-bound test, while the search for the matching MD5 hash is an I/O-bound test. Therefore, the benchmark is not limited to one type of operation.

Note that it is in fact possible to select any other operations for the Map-Reduce benchmark and they could be easily implemented as well. The search for the plain text that was used to generate a supplied MD5 hash is selected because of further important aspects.

The structure of the input data is well defined. This implies that all of the blocks of input data are the same length, which does not introduce any jitter into the duration of the computation of that particular unit of data. Moreover, the input data can be well divided into evenly sized *Chunk*s between the *Mapper*s. This allows using evenly sized work units across the cluster. All in all, no *Worker* is impeded in any way.

Next, the computation of the MD5 hash function is a non-trivial operation requiring many CPU cycles, yet the lower bound on the number of cycles is known. More detailed discussion about the MD5 algorithm is presented in Subsection 7.2.1 on page 28. The operation is well understood, analyzed and implemented on all of the hardware used throughout this text.

The benchmark's complexity grows exponentially with increasing length of the input data. To make sure the benchmark is still computationally viable and to avoid wasting precious computational resources, the length of the input data is fixed to *28 bit*s throughout the text. While this number does sound small, the result is a set of input data which consists of exactly 2^{28}, i.e. over 256 Million (key, value) pairs. Given that the representation of a *28 bit* number requires *4 Bytes* of memory, the net size of the raw input data is is $4B * 2^{28} = 1GiB$.

Based on early observations, this size of the input data is already sufficient to make the benchmark non-trivial on all tested platforms. It is also computable on all of the test systems in a reasonable amount of time.

5.2 The benchmark in Map-Reduce context

The input data set for this benchmark is a list of 2^{28} (key, value) pairs, as expected by the Map-Reduce algorithm. In this particular case, the input *key* is ignored and the only part of the input (key, value) pair used further is the *value*. The value is a 4 Byte long binary string, of which the bottom 28 bits are used. Since the input data contain all possible 28 bit long binary strings, it is easy to observe that the input data can be divided into comparably sized *Chunk*s in between the *Mapper*s.

Let I be the set of input (key, value) pairs, the *Map()* function implements a transformation as follows:

$$\forall(key, value) \in I : Map((key, value)) \rightarrow (value[0], (value, MD5(value)))$$

The data passed to the *Intermediate* stage is in the rather obscure format of $(value[0], (value, MD5(value)))$ pairs. The intermediate *key* is the lowest significant byte of the original input value and the intermediate *value* is a concatenation of the whole original value and an MD5 hash of the value. It is indeed true that the intermediate format contains one byte of redundant information.

The reason for such an obscure format is that the default *Combiner* groups together the (key, value) pairs emitted by the *Mapper*, which share the same intermediate *key*. By using the one byte of *value*, i.e $value[0]$, as an intermediate *key*, the *Combiner* can generate only up to 256 possible distinct groups. Furthermore, the sum of the number of elements in each of the 256 distinct groups across all *Nodes* in the cluster is the same, because each number is represented the same amount of times in $value[0]$.

The sum of size of all values associated with a single group across all *Node*s in the cluster is $(4B + 16B) * \frac{2^{28}}{256} = 20MiB$. The size of the particular *value* is 4 bytes and the size of the MD5 hash is 16 bytes. It is explicitly not necessary that all of the values for a single group end up in the same *Shard* or are produced on the same *Node*. Yet, the work units exchanged from *Mapper*s to *Reducer*s are well defined.

Let S be all *Shard*s assigned to Reducer and let H be the target MD5 hash supplied by the user, then the *Reduce()* function implements a transformation as follows:

$$\begin{aligned}\forall s \in S : &\forall (key, Values) \in s : \forall (value, MD5(value)) \in Values : \\ &Reduce(key, Values) \rightarrow Results \;\&\& \\ &((value, MD5(value)) \in Results \Leftrightarrow MD5(value) = H)\end{aligned}$$

The resulting data from the *Reduce()* function are zero or more pairs of the form $(value, MD5(value))$, where the *MD5(value)* matches the MD5 hash provided by the user and the *value* is the matching input string from which the MD5 hash was generated.

6. Measurement methodology

This text focuses on increasing the power efficiency of a Map-Reduce cluster, it is therefore necessary to establish a method for measuring the power consumption in proportion to the amount of computation done by the cluster.

6.1 Quantifying efficiency

There are two variables at play here, which go against each other, one being the power consumption per unit of time and the other being the amount of transformations done by the cluster per unit of time.

The performance of the Map-Reduce cluster can be measured by the amount of transformed (key, value) pairs per unit of time. The measurements in the rest of this text are conducted with a 1 second stepping, therefore the unit of time selected here is also a second. The amount of transformed (key, value) pairs by a cluster in one second time is henceforth denoted *Transformations per Second*, abbreviated *TpS*.

The *TpS* is effectively the amount of work done by the cluster per second. The amount of work done by an electrical circuit per unit of time is measured in *Joule* [*J*] and one Joule is defined as $1J = \frac{1W}{1s}$, which is formally written as $W = \frac{P}{t}$

Correlating the amount of work done by the cluster per second (TpS) with the amount of electrical work required to operate the cluster for one second (W) according to Figure 6.1, the coefficient TpW is obtained.

$$TpW = \frac{TpS}{W[J]} = \frac{\frac{T[transformations]}{t[s]}}{\frac{P[W]}{t[s]}} = \frac{transformations}{P[W]}$$

Figure 6.1: Formula for calculation of the TpW ratio

The coefficient is henceforth called *Transformations per Watt*, abbreviated *TpW*. The coefficient normalizes the amount of transformations, which can be achieved on different machines by consuming the same unit of power. This in turn makes this coefficient a good fit for comparing the power efficiency of completely different hardware.

6.2 TpW sampling

Since it is not easily possible to discern how much power was used to calculate a particular *Chunk* of input data, for each test in this text, the power consumption is sampled for the entire duration of the Map-Reduce job and divided by the duration of it to yield the average *TpW* of the entire job.

This average *TpW* value is used to compare the efficiency of different types of hardware at processing the same Map-Reduce Job. The node exposing the highest average *TpW* is then the most power efficient one.

For the benchmark used in this text, the average *TpW* is determined from the measurement using the formula in Figure 6.2, where t_{total} is the total duration

$$TpW = \frac{2^{28}}{\sum_{t=0}^{t_{total}} P_t}$$

Figure 6.2: Formula for calculation of the average TpW ratio

of the Map-Reduce computation on the worker node and P_t is the power consumption of the node during the t-th second after the start of the Map-Reduce computation.

6.3 Measurement instrumentation

Initially, it was expected that a common UPS would be able to provide accurate power consumption data for both the reference x86 system and the ARM systems, yet the evaluation of the characteristics of the UPS showed this expectation to be unrealistic. In this particular case, the UPS model is an APC Back-UPS Pro 1500.

An experimental measurement shows that the consumption of either of the available ARM systems is in the range of 5-10 W , while the consumption of the reference x86 system is in the range of 100-300 W. The resolution of the particular UPS is 1 W. The maximum absolute error is thus 0.5 W.

Considering a mean value of the power consumption of either systems, the *relative error* for the reference x86 system is in the order of $\delta_{x86} = \frac{0.5W}{200W} = 0.0025 = 0.25\%$ and for either of the ARM systems is $\delta_{arm} = \frac{0.5W}{7.5W} = 0.0667 = 6.67\%$. This clearly shows that the low resolution of the UPS is suitable only for measuring the reference x86 system, but has a grave impact on the result of the ARM systems. Measuring the ARM systems with a UPS would not yield enough precision for the results to be reliable.

The strategies for measuring the power consumption of the reference x86 system and the ARM systems therefore differ.

6.3.1 The reference system measurement

As discussed above, the UPS is used throughout the rest of this text to determine the consumption of the reference x86 system. A tool called `apcupsd` is used to read the immediate power consumption data from the UPS via the UPS's USB interface for further processing.

6.3.2 The ARM systems measurement

As the ARM systems cannot be reliably measured using the UPS, a different method was needed to yield more precise results at these levels of power consumption.

The requirement is to be able to measure power consumption of an ARM system in proportion to time. An oscilloscope is a device which displays a voltage difference in proportion to time. An easy circuit was constructed, in which it is possible to measure a voltage drop on a resistor. Such voltage drop is a value which can be monitored using the oscilloscope in proportion to time. By applying

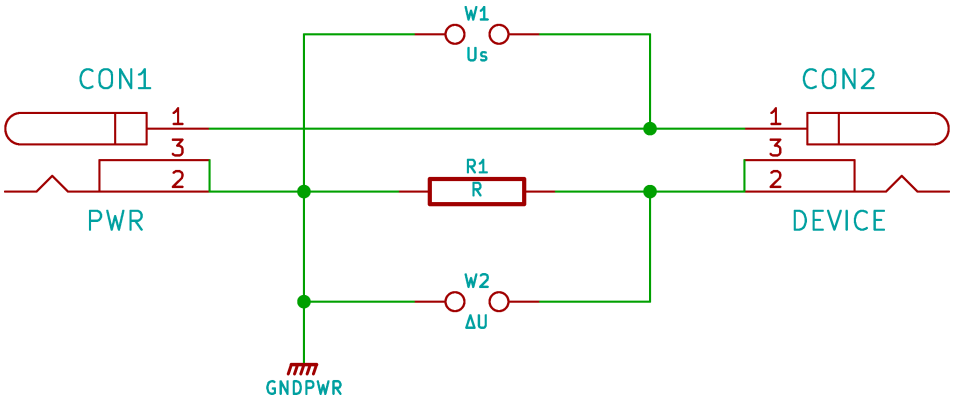


Figure 6.3: Schematic of the measurement helper circuit for ARM systems

Ohm's law and Kirchhoff's laws, it is possible to calculate the power consumption of a circuit from aforementioned voltage drop.

The schematic of the helper instrument for measuring the power consumption of the ARM systems is presented in Figure 6.3. The DENX MCVEVK system is supplied by 9.28 V and the Kosagi Novena is supplied by 16.30 V from a power supply. This supply voltage is provided on the connector `CON1`. The device itself is connected to connector `CON2`. In the default case, where the device is connected directly to the power supply, the resistance of the resistor `R1` is zero.

There are a couple of requirements to measure a power consumption of the device without introducing incorrect results in the process. The most important requirement is that the voltage of the power supply must not fluctuate under load. A laboratory power supply was used in case of this text to provide a strong and stable power source instead of the regular wall plug. It will shortly become obvious why the stability is important.

The supply voltage is measured on pads `W1` and is henceforth denoted as U_s. The U_s must be measured before any other measurements can be conducted to determine it's stability. The U_s must be measured twice, once without the device connected to determine the maximum U_s and once with the device connected to determine how much the U_s dropped. In case of the laboratory supply used in this text, the U_s drop is marginal and can be safely ignored without obscuring the results.

The resistance of the resistor `R1` must be very low, otherwise the voltage drop on the resistor might cause the voltage supplied to the measured device to be insufficient. A higher resistance of the resistor `R1` would also cause heating of the resistor itself, since the losses would be transformed into heat and such behavior is unwanted. An 1Ω resistor was used for the measurements in this text. The voltage drop on the resistor `R1` is measured on pads `W2` and is henceforth denoted as ΔU.

$$\sum_{k=1}^{n} V_k = 0$$

Figure 6.4: Kirchhoff Voltage Law (KVL)

The Kirchhoff Voltage Law (KVL) presented in Figure 6.4 states that the sum of voltages in the entire circuit equals zero. In case of this particular circuit, this

means that $U_s = \Delta U + \Delta U_{device}$.

$$\sum_{k=1}^{n} I_k = 0$$

Figure 6.5: Kirchhoff Current Law (KCL)

The Kirchhoff Current Law (KCL) presented in Figure 6.5 states that the sum of electric currents flowing into any junction of a circuit is equal to the sum of electric currents flowing from that junction.

$$I = \frac{U}{R}$$

Figure 6.6: Ohm's law

Finally, Ohm's law in Figure 6.6 shows the relation between resistance, voltage and current.

$$P = UI = \frac{U_{device}\Delta U}{R} = \frac{(U_s - \Delta U)\Delta U}{R}$$

Figure 6.7: Relation between P and ΔU

The relation between the electrical power P and the voltage drop ΔU on resistor `R1` is presented in Figure 6.7. The second equation comes from the Ohm's law and KCL, the third equation is just an application of the KVL onto U_{device}. The variables U_s and R are known, therefore there is a direct relation between P and ΔU and the ΔU is a variable measured by the oscilloscope in proportion to time.

Should the U_s voltage fluctuate with changing load of the circuit, both the $U_s - \Delta U$ and ΔU would also change and the results would become useless. It is therefore indeed important to use a stable power supply.

On a final note, it might come as a surprise that the resistor `R1` is connected past the measured device. The placement of the resistor does not have any impact on the measurement because of the KCL. Yet, placing the `R1` past the device is necessary because of the oscilloscope. Unless the scope has a differential probe, the regular scope probe has a sampling input and a ground lead. It would not be possible to attach the ground lead anywhere if the `R1` was connected before the device, since connecting the ground lead would immediately cause a short circuit and damage the power supply.

6.4 Processing the data

Unless stated otherwise, every measurement for each configuration from this point on was conducted five times. While sampling every case five times does indeed look like a low amount of samples, the standard deviation of the results is always low, which indicates there is not much fluctuation in the measurements.

There are two sources of raw data, which must be processed. The first source of raw data is the log coming from the Disco Map-Reduce framework, while the second source is the measurement of power consumption. All of the data,

intermediate results and scripts used to do measurements are present on the attached CD in the *measurements/* directory.

The sampled data are processed using a very basic helper script called *proc.sh*, which extracts the necessary information from both the Disco log and the power consumption data and executes another helper script implemented in R 3.1.1 called *stat.R* to generate the necessary statistics over these extracted data. The following data are extracted and used for statistics:

1. Duration of Map, Shuffle, Reduce stage and the total duration of the job
2. Number and type of tasks running on the cluster in 1 second step
3. Power consumption data in either UPS or scope format

Extracting the timing information is trivial, since this involves parsing the Disco log and this is what the Unix tools were originally meant for anyway. The timing data granularity is 1 second. The measurement of power consumption is much trickier.

In case of the reference x86 system, a script[1] is used to control the entire process of sampling all the data during the benchmark. The script calls the aforementioned `apcupsd` once every second and adjusts the output slightly to select only the power consumption. The result is a list of values, each representing the power consumption of the reference x86 system in 1 second intervals. Data in this format can already be inspected in relation to the timing data mentioned above.

In case of the ARM systems, a script[2] is used to control the entire process of sampling all the data during the benchmark as well. In this case, the script calls the `utc`[3] tool to retrieve raw data from the scope.

The scope is configured with 1 second time base, 100mV step and 0mV offset. Every sample retrieved from the scope spans 12 seconds and contains a total of exactly 6000 data points. The sampling is done once every 10 seconds and the trailing 2 seconds of data are discarded, since those data are in the subsequent 10 second sample. The 10 second sample thus contains 5000 data points, so 1 second consists of 500 samples.

The *stat.R* script first transforms every data point from scope format to the voltage difference on the resistor and then to power consumption according to the formula Subsection 6.3.2 . The R script then calculates averages for every 500 data points, resulting in a data in the same format as the UPS, i.e. one datum per second.

With the data in correct format for both x86 and ARM systems, the *stat.R* script then calculates average duration of Map, Shuffle, Reduce stage, average duration of the whole job and average TpW from all five measurements. A standard deviation is calculated to indicate stability of the data. Each measurement result presented in later chapters is thus in the format:

$$(value \pm standard\ deviation)$$

[1] *measurements/bin/x86-getwatt.sh* on the CD
[2] *measurements/bin/arm-getwatt.sh* on the CD
[3] *measurements/bin/utc* and *measurements/src/uni-t-control* on the CD

7. Computation on ARM-FPGA

7.1 ARM-FPGA characteristics

The Altera SoCFPGA Cyclone V hardware has characteristics which must be taken into consideration before such a system can be used to do any computation.

Likely the most surprising feature of the ARM-FPGA system is the clock speed of the FPGA, which is by no means fast. According to the DENX MCVEVK documentation [16], the FPGA reference clock is running at 50 MHz on this system, whereas the CPU clock is running at 800 MHz. It does therefore come as a surprise how such a slow device can help process large amounts of data quickly.

An established trend in the computer industry shows that instead of scaling the frequency, it is often possible to introduce parallelism into the design. The FPGA is capable of doing just that – process data in parallel at lower clock speeds. Not only that, but the FPGA is capable of doing the parallel processing on a truly massive scale.

Moreover, because the hardware that is programmed into the FPGA is completely custom tailored for the particular computation, the hardware is optimized in such a way that not a single clock cycle is wasted.

7.2 Implementing MD5 on FPGA

To accelerate the MD5 on FPGA, a suitable MD5 IP block must be programmed into the FPGA. Since the target for the MD5 IP block is to be used in the Map-Reduce context and thus compute large amounts of the MD5 hashes on independent blocks, it is possible for the implementation to compute multiple blocks in parallel.

7.2.1 MD5 algorithm detail

The MD5 hash calculation starts with two inputs, zero or more blocks of data, which are provided by the user and an initialization vector, IV for short. For the first block of the MD5 computation, the IV is defined in the *RFC 1321, The MD5 Message-Digest Algorithm* [8] document, in particular *Section 3.3 Step 3. Initialize MD Buffer* .

The length of one MD5 block is 447 bits. Since the length of the values for each (key, value) pair in the benchmark is 28 bits, which is much shorter than 447 bits, the text does not deal with MD5 hashes computed from more than one block.

It is nonetheless trivial to extend an MD5 hash computation to more than one block, both in software and in hardware. In the software case, the IV for the subsequent block of computation is the hash of the previous block. In the hardware case, the MD5 units are simply chained one after the other or the data from the MD5 unit are fed back into the input of the MD5 unit.

The computation of a single MD5 block takes exactly 64 rounds of a loop. The loop can be decomposed into four different parts, depending on how far the computation progressed throughout the loop. During every round of the loop, the

data are subjected to basic binary and arithmetic operations, which are detailed in *RFC 1321, The MD5 Message-Digest Algorithm* [8], *Section 3.3 Step 4. Process Message in 16-Word Blocks*. The hash computation is completed after 64 rounds of the loop.

It is important to note that there is an explicit interdependence between the data in every round of the loop, therefore it is not possible to parallelize the execution of the loop itself.

7.2.2 MD5 hardware detail

The MD5 implementation can be broken into three parts:

1. MD5 computational core
2. Data adaptation layer
3. Data transport engine

The MD5 core is responsible for transforming a block of data, which is presented on a set of wires within the FPGA and to present the result on another set of wires in the FPGA.

The Data adaptation layer is responsible for converting the data format between the CPU and the FPGA, so that both can operate on data in their native format.

The Data transport engine is responsible for moving data between the CPU and FPGA. The engine is usually implemented in a form of DMA controller.

The implementation of the MD5 algorithm is trivial and since a suitable MD5 core was already available under a useable license, this text uses an MD5 core[1] borrowed from OpenCores, originally written by John Leitch [19].

The property of this particular MD5 core, which has both an up- and downside, is that it uses pipelining to speed up the calculation of MD5 hashes. The MD5 core implementation literally unwinds the MD5 computation loop within the FPGA in such a way that all stages of the loop exist at the same time.

The block of data, which is presented on the input wires of the MD5 IP block is loaded into the MD5 IP block on the 0-th clock edge. At this point, the block of data enters the first round of the MD5 loop. On every subsequent clock tick, the block of data moves forward in the pipeline. On the N-th clock tick, an operation which corresponds to the N-th iteration of the MD5 loop is executed on the block of data and the block moves to N+1-th position in the pipeline. After 64 clock ticks, the block of data is presented on the output wires of the MD5 block.

It is important to note that it still takes 64 ticks to compute a single MD5 hash. It is also important to notice that after the first 64 clock ticks, the MD5 IP block will present one MD5 hash on the output wires on every further tick. Finally, once there are no more input data blocks, to avoid loosing the hashes for the last 64 blocks of input data, the user must do 64 more clock ticks, in which the MD5 IP block flushes the pipeline and presents the remaining 64 hashes.

Therefore, the downside of the pipelined approach is that there is a 64 ticks lead setup time, in which no results are output and there is a 64 ticks shutdown

[1] `fpga/src/hw/0001-Add-streaming-MD5-core.patch` on the CD

time at the end of computation. The upside is that in between these two time frames, the MD5 IP block generates 1 MD5 hash per clock tick. It is therefore beneficial to compute as much data on the MD5 IP block in one long run as possible to mitigate the setup overhead.

The above text still dealt with the raw MD5 IP block. It is not possible for the CPU to interface with the MD5 IP block in the FPGA directly. Therefore, I implemented a glue layer[2], which contains the Data transport engine and does mild adaptation of the data format between the CPU and the FPGA. The full FPGA project source is on the attached CD in `fpga/src/hw/fpga-md5` directory.

The glue layer consists of two DMA engines and a thin adaptation layer for the data. One of the DMA engines moves the MD5 blocks from RAM into the FPGA, where they are processed using the MD5 IP block. The other DMA engine moves the resulting MD5 hashes back from the FPGA into the RAM.

The pipelined MD5 IP block described above does not implement the addition of the IV to the resulting MD5 hash, which is done at the end of *Section 3.3 Step 4. Process Message in 16-Word Blocks* of the *RFC 1321, The MD5 Message-Digest Algorithm* [8] . The pipelined MD5 IP block only unwinds the MD5 computation loop. As this addition is required to receive a valid MD5 hash, the glue layer does this addition before the hash produced by the MD5 IP block is sent via DMA into CPU's memory.

On a final note, the glue layer also does endianness adjustment between the FPGA and the CPU, so that the data are in the correct format for both.

The DMA engines used in the adaptation layer are configured such that the upper limit on the amount of data, which can be transferred in a single run is 16 MiB. Since the MD5 block is 64 Byte long, the maximum amount of MD5 blocks, which can be processed at one run of the DMA engine is 2^{18} blocks. Note that this number is important and will be used a lot throughout the rest of the text, especially when testing the FPGA performance. The minimal duration of the computation of a full 2^{18} of MD5 blocks on the FPGA, in ideal conditions, is therefore:

$$t = \frac{64+2^{18}}{50*10^6[MHz]} = 0.005s$$

The duration of the computation of the entirety of the input data for the benchmark is $t_b = \frac{2^{28}}{2^{18}} * t = 5.2s$. Theoretically, the computation of a large amount of data on the FPGA is thus extremely fast.

7.3 Interfacing the FPGA from Linux

There are multiple steps which must be executed in the correct order to use the FPGA at all. From a high-level perspective, these are:

1. Program the FPGA with bitstream
2. Connect communication bridges between CPU and FPGA
3. Perform computation on the FPGA

[2] `fpga/src/hw/0002-Add-MD5-core-adaptation-layer.patch` on the CD

The first step is to program the FPGA such that the hardware is synthesized in the FPGA. The blob of data, which represents the hardware in the FPGA is called a `bitstream`. From the Linux perspective, this operation is a single command that must be executed as a superuser.

```
$ cat md5.rbf > /dev/fpga0
```

It is noticeable that programming the FPGA is not fast. A brief test using the `time(1)` command shows the following result:

```
$ time cat /root/md5.rbf > /dev/fpga0

real    0m0.488s
user    0m0.000s
sys     0m0.416s
```

This means programming of the FPGA takes about 0.5 seconds. Comparing this to the duration of a full computation of 2^{18} sized block of input data for the FPGA, taking 0.005 seconds as explained in previous section, it is obvious that it is not feasible to re-program the FPGA for each block of data which is to be computed on the FPGA. Unlike a CPU, where the tasks can be easily swapped back and forth, the FPGA thus requires a different treatment. The Section 8.3 on page 39 outlines how this problem was solved.

The communication bridges between the CPU and FPGA facilitate all communication between the CPU and FPGA. They allow the CPU to access resources synthesized in the FPGA and vice versa. The FPGA can also access the RAM via these bridges. The bridges are enabled using the following commands, which on the kernel level translate into a write into a CPU register. The duration of execution of these commands is close to zero and thus negligible:

```
$ echo 1 > /sys/class/fpga-bridge/hps2fpga/enable
$ echo 1 > /sys/class/fpga-bridge/fpga2hps/enable
$ echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable
```

Finally, the user performs the computation accelerated by the IP blocks synthesized in the FPGA. With the bridges enabled like described above, the user can access the registers of the IP blocks loaded in the FPGA. These registers are mapped into a part of the CPU's address space. In this particular case, the registers that are synthesized in the FPGA are used to control the two DMA engines in the MD5 data transport engine. As mentioned earlier, the FPGA can also access the contents of the RAM, which is what makes a DMA transfer initiated by the FPGA possible. Detailed documentation for the DMA engines can be found at *Altera Wiki* article *Modular SGDMA* [21].

The only real obstacle here is how to allow a regular user operate the FPGA, so that the FPGA can be used in the Map-Reduce computation. The hardware is typically controlled by the operating system kernel, the Linux kernel in this particular case. The requirement is that the user can operate the registers synthesized in the FPGA and the user can prepare data in the RAM for the DMA engines in the FPGA.

An obvious solution is to use loadable kernel modules supported by the Linux kernel. Therefore each of the Map-Reduce jobs could carry an FPGA bitstream with it and a Linux kernel module to control the hardware. The obvious problem is that the user is responsible for providing a correct Linux kernel module.

The user must therefore be knowledgeable about the register mapping of the target *Node*. The user is also responsible for writing the kernel module in such a way that it won't have any adverse effects on the target *Node*. To make matters even worse, the user must build the kernel module for a particular kernel version. This solution clearly has limited use in the real world.

Instead, the implementation in this text uses a different approach, built from two components, the `generic-uio` and *Contiguous Memory Allocator*, *CMA* for short, both of which are explained below.

The Linux kernel supports a `/dev/mem` device that can be used by privileged user land tools to access any address in the system's memory space. The `generic-uio` is an abstraction of `/dev/mem` in such a way that on the kernel level, it allows exporting only a subset of the whole system's memory space. The approach taken in this text uses the `generic-uio` to export exactly the area of the CPU's address space, which is dedicated for accessing the control registers synthesized in the FPGA.

In order to use it, the user `open(2)`s a `/dev/uio*` device and does an `mmap(2)` operation on it's file descriptor. This maps the file contents into the process address space and by doing memory accesses on these addresses, the user accesses the control registers of the IP block in the FPGA. The benefit on this setup is that the user does not need to know anything about the address space layout of the system. The user only accesses the control registers by doing memory accesses to an address relative to the start of the `mmap(2)`'d area.

The above caters for the part where the user must operate the control registers in the FPGA. The user also needs a way to prepare data in the RAM so they can be picked by the DMA engines in the FPGA. The problem here is twofold. The DMA engines typically cannot read data from arbitrary memory locations, but require the memory location address to be aligned. The other problem is that the DMA engine accesses the RAM directly, while the CPU accesses the RAM via multiple levels of CPU caches.

Both of the above problems are handled by a small Linux kernel module[3], which was developed for the purpose of this text and which uses the *CMA*. The reason for using *CMA* in this development is that the Linux kernel memory allocator doesn't allow allocating huge amounts of continuous memory, but the *CMA* allows exactly that. The approach here uses the *CMA* to reserve 256 MiB of system memory. The kernel module called `fpga_cma` then allocates this amount of memory and creates a device node. The user can again `open(2)` and `mmap(2)` this device node to access this *CMA* memory from user space.

The property of this memory is that it is automatically page-aligned. The size of a page is 4 kiB on ARM, which means the start address of the `mmap(2)`'d area is aligned to 4 kiB in the physical RAM. This alignment is sufficient for the used DMA engines.

[3] `fpga/src/sw/linux/0001-misc-Add-trivial-CMA-module-for-the-FPGA.patch` on the CD

To mitigate the problem with CPU caches, the `fpga_cma` module marks the whole *CMA* memory area as non-cachable. All of the accesses the user does on this area will be done directly on the physical RAM. From the DMA engine side, the data the user wrote into the RAM are immediately visible; from the user side, the data which were written into the RAM by the DMA are immediately visible to the user.

8. Benchmark implementation

Integrating the MD5 benchmark into the Disco Map-Reduce framework is straightforward when the benchmark is implemented using plain software. There are additional special considerations when the benchmark is implemented in hardware.

8.1 Benchmark job design

The benchmark job is a classical Map-Reduce job, which implements both Map() and Reduce() functions and operates on data placed on a distributed file system.

A schematic of a typical Disco Map-Reduce job is presented in Figure 8.1. The user provides an implementation of the Map-Reduce job to the Map-Reduce Master and pushes the input data to the Disco Distributed Filesystem, DDFS. The Disco Master distributes the Map and Reduce tasks to the Workers in the cluster. Once finished, the Reducers report their partial results back to the Disco Master, which in turn presents the result back to the user.

8.1.1 Map-Reduce job design

The Map-Reduce job in Disco is often implemented in a single file and takes only a few lines of code. There are three main components of the Disco job. The full source code for the benchmark is available in the file `disco/job/md5-bench.py` on the attached CD. The file contains additional boilerplate code, so the text below presents the relevant sections only. First is the snippet of a Disco Job main function in Figure 8.2 which spawns the job and presents the results:

The job is spawned by creating a `Job` object and executed by calling it's `.run()` method. The `input` argument specifies the location of the input data for this Map-Reduce job. In this particular case, the `tag://...` prefix specifies that the data are located on the DDFS and identified by the DDFS *Tag*. More details on DDFS can be found in Section 8.1.2. The `map` and `reduce` arguments specify the Map() and Reduce() functions of the job respectively.

The `map_reader = chain_reader` argument is important. It replaces the default input reader for the Map() function with one capable of reading data in the format provided by the DDFS. Without this argument, it would not be possible for the Job to read data from DDFS.

The trailing for loop prints the results of the Map-Reduce computation.

Next is the Map() function. The `soft_map(kv, param)` function is called with two arguments. The `kv` argument contains a single (key, value) pair for which the function computes a transformation, the `param` argument is an optional set of arguments passed from Job() via the `params` argument. The Map() function uses `yield key, value` to return zero or more (key, value) pairs.

Finally the Reduce() function. The `soft_reduce(iter, params)` function is again called with two arguments. The `param` argument has the same meaning as in the Map() case. The `iter` argument is an iterator over the provided (key, list of values) pairs. Same as in the Map() case, the `yield key, value` is used to return (key, value) pairs.

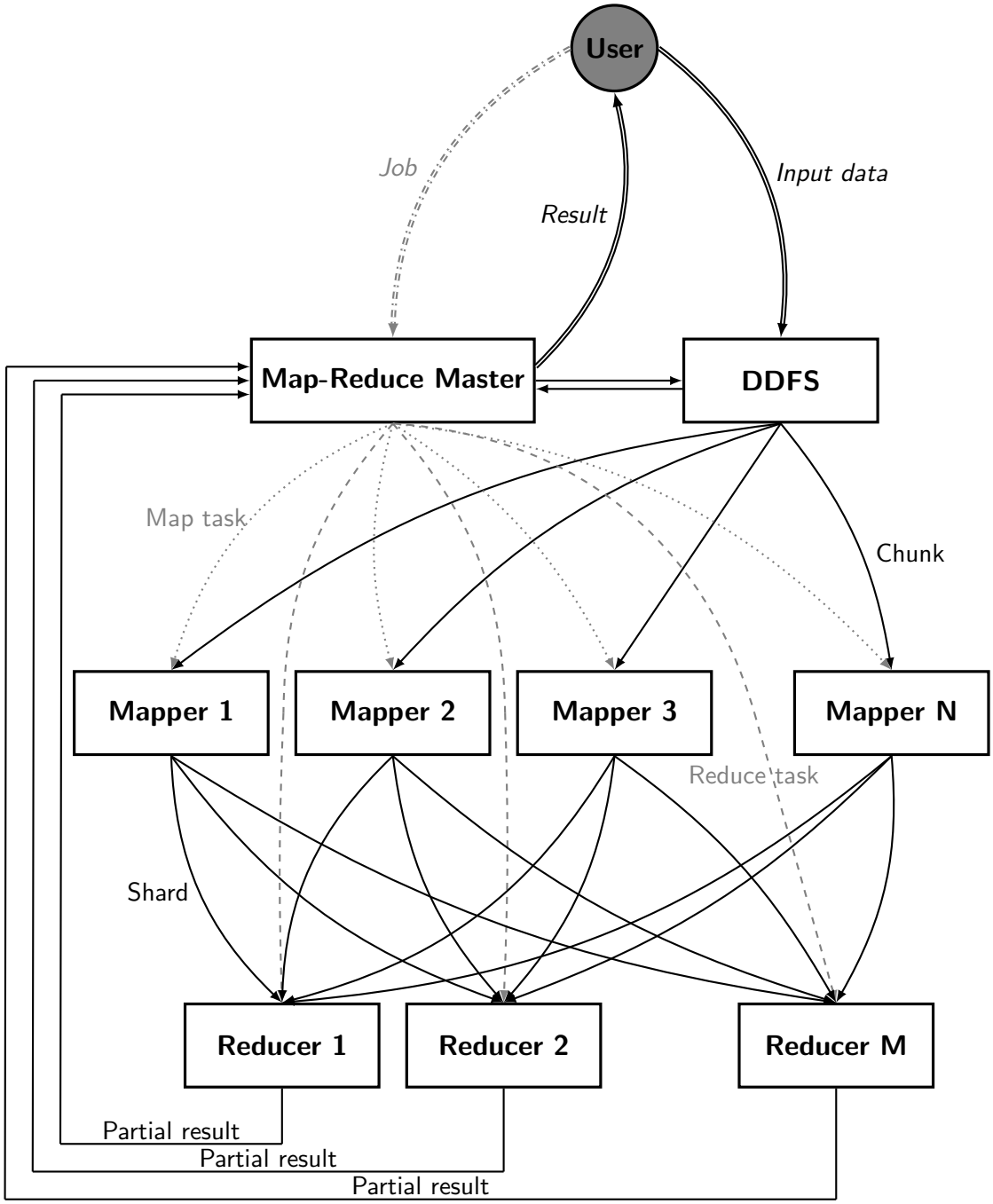


Figure 8.1: Flow in the Disco Map-Reduce job

8.1.2 Remote storage

In case of the Disco Map-Reduce framework, there are multiple options from where the input data for a Map-Reduce job can be sourced. This text uses the Disco Distributed Filesystem, the DDFS, to store the input data for the Map-Reduce job. The decision to use the DDFS instead of other options was made because the DDFS is recommended input by the Disco project documentation.

The entire payload pushed into the DDFS is identified by a *Tag* and has zero or more *Blob*s associated with it. Each Blob contains part of the payload identified by the *Tag*. The number of *Blob*s associated with a *Tag* is also the upper limit on

```
from disco.core import Job, result_iterator
from disco.worker.classic.func import chain_reader
#...
if __name__ == '__main__':
    rhash = '096822475dd517c1089f4b58c7f346ef'
    job = Job().run(input=["tag://data:md5:bin1"],
                    map=soft_map, reduce=soft_reduce,
                    params={'csum': bytearray.fromhex(rhash)},
                    map_reader = chain_reader);
    for key, val in result_iterator(job.wait(show=True)):
        print(key, val)
```

Figure 8.2: Disco Map-Reduce job main function snippet

the number of *Mapper*s which can operate on input data identified by that *Tag*. Each *Mapper* has one *Blob* associated with it. Notice that the *Blob*s used by the DDFS match the properties of *Chunk*s from the generic Map-Reduce description in Chapter 2 on page 9.

For convenience, the DDFS supports automatic splitting of the input data into evenly sized *Blob*s. The size of the *Blob* can be configured via the command line interface of the DDFS client.

For the purpose of the benchmark used in this text, another feature of the DDFS was used. It is possible to provide an input data reader implementation for the DDFS client replacing the original input data reader. The input data reader is a piece of python code that parses the input file to be pushed into the DDFS and emits (key, value) pairs, which are then stored into the DDFS.

The benchmark specified in Chapter 5 on page 20 defines the format of (key, value) pairs used as an input data such that the key is irrelevant and the value is a 28-bit long binary string used as plain text input for the MD5 function. The replacement input data reader therefore generates all possible 28 bit long binary strings and emits them as (key, value) pairs, where the *key* has a value 'k' and the *value* is the generated binary string. A simple python function implementing the input data reader by generating such a full range of (key, value) pairs for the DDFS is presented in Figure 8.3 .

```
def soft_chunk_reader(stream, size, url, params):
    import struct
    for i in range(0, 2**28):
        yield "k", struct.pack(">i", i)
```

Figure 8.3: DDFS input data reader implementation, 1 plain text per key

The command presented in Figure 8.4 executes the `soft_chunk_reader()` function and loads the data into the DDFS.

The adjustment to `PYTHONPATH` is necessary so that Python can find the file and function that is executed by the DDFS client as the input data reader. This function is specified by the `--reader` argument of the `ddfs chunk` command.

```
$ PYTHONPATH=$PYTHONPATH:md5/ ddfs chunk -S 4 data:md5:bin1       \
                                      /dev/null                   \
                                      --reader md5.soft_chunk_reader
```

Figure 8.4: DDFS input data reader execution, 1 plain text per key

The `ddfs chunk` command loads the input file into the DDFS and assigns a `data:md5:bin1` *Tag* to the data. Since all the input data are generated by the input data reader and no external data are necessary, the source file `/dev/null` is used. This is legal, since the only operations done on this file would be open and close, no read operation are executed.

As was mentioned before, the DDFS supports splitting input data into evenly sized *Blob*s. The `-S` parameter configures the size of the *Blob* and in this case the size is set to 4 MiB. This value was derived empirically and results in the generation of 150 *Blob*s, which assures that there are enough *Mapper*s available when doing the measurements on a cluster.

8.1.3 Map-Reduce job execution

The Map-Reduce job is pushed into the Disco cluster by executing the file implementing the job. As the file implementing the MD5 benchmark is called `md5-bench.py`, the job is sent to Disco using the following command:

```
$ python md5-bench.py
```

The situation on a *Node* in the cluster after a Job was executed is presented in Figure 8.5. Each of the boxes represent one running process on that *Node*. It is important to note that in the general case, there are multiple `Mapper`s and multiple `Reducer`s running on the *Node* at the same time. It is also important to clarify that all the `Mappers` associated with one Job processes output into single `Combiner` and *Shuffle* produces *Shards* from the same bulk of data. Note that the rest is the same as in the generic Map-Reduce case as explained in Chapter 2 on page 9.

8.2 Software benchmark job

The software implementation of the benchmark job is trivial and is done entirely in Python within the realm of the Disco Map-Reduce framework itself. Both the Map() and Reduce() functions are very straightforward and are presented in Figure 8.6.

The `soft_map()` and the `soft_reduce()` functions map directly to `Map()` and `Reduce()` functions in Figure 8.5 respectively.

The `Map()` function first extracts the value from the (key, value) pair passed in via the `kv` parameter. The next line calculates the MD5 hash of the value and the last line emits a (key, value) pair in the format $(value[0], (value, MD5(value))$. This matches the format specified in the benchmark in Section 5.2 on page 21. It is now clear that this part of the benchmark is CPU bound, as was already stated in Section 5.1.

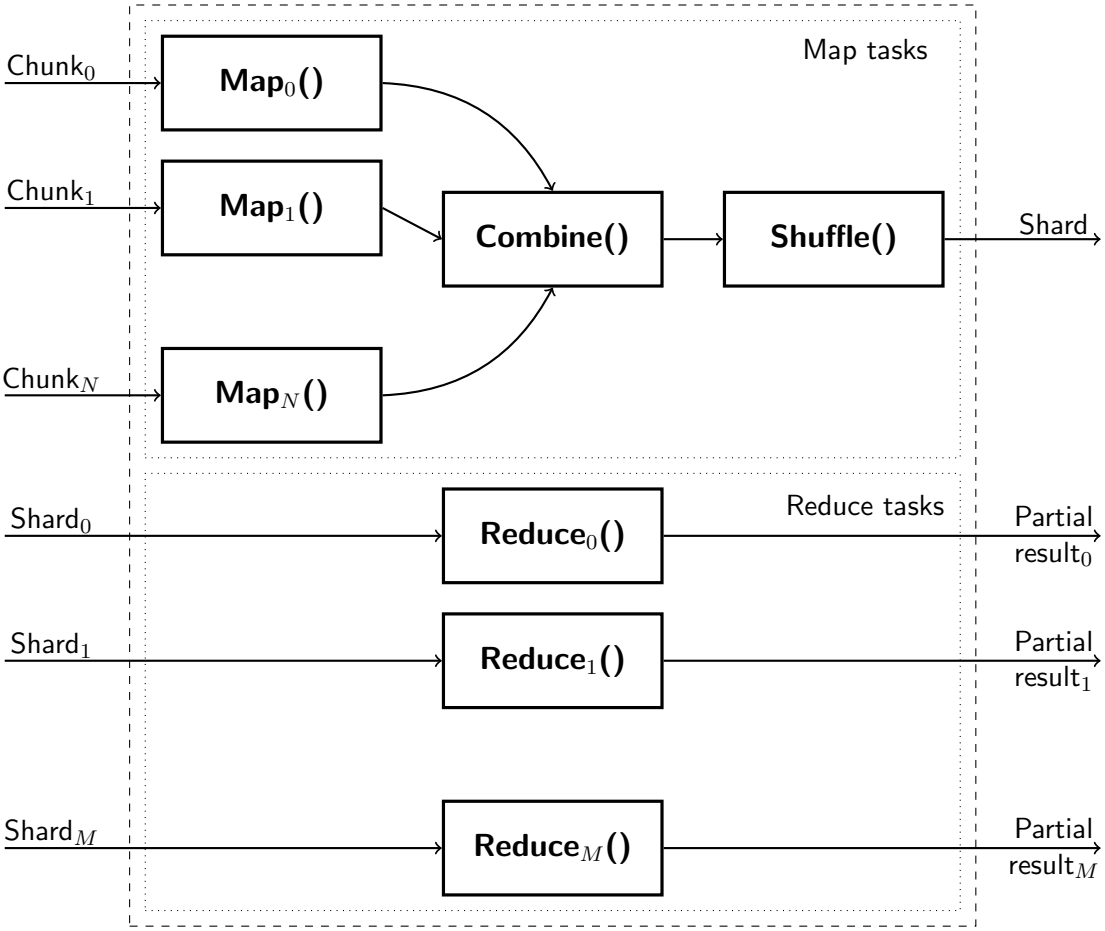


Figure 8.5: Processes on a single Node in Disco cluster

```
def soft_map(kv, params):
    val = kv[1]
    hsh = hashlib.md5(val).digest()
    yield val[3], val + hsh

def soft_reduce(iter, params):
    from disco.util import kvgroup
    for valzero, pairs in kvgroup(iter):
        for pair in pairs:
            if (pair[4:] == params['csum']):
                yield pair[0:4], pair[4:]
```

Figure 8.6: Software Map and Reduce implementation, 1 plain text per key

It might come as a surprise that the `val[3]` is used to extract the LSB from the binary string. This is correct, since the MSB of the represented value is at position `val[0]` because both the ARM and the reference x86 systems are *Little Endian*.

The `Reduce()` function iterates over all provided pairs of the (key, list of values) using the `iter` iterator. Notice that the `pairs` argument is again a list of values associated with the `valzero` key. Remember that the `valzero` is the LSB

of the initial data, as was explained in Section 5.2 on page 21. The implementation iterates over the `pairs` list, whose entries have the format $(value, MD5(value))$ and tests if the particular `pair` contains the target MD5 hash. If the matching MD5 hash is found, that particular `pair` is emitted. As was pointed out in Section 5.1, this part of the benchmark is I/O-bound.

8.3 Hardware benchmark job

The hardware assisted implementation of the benchmark job is much more complex. There are two orthogonal problems that must be solved.

The first problem is controlling the FPGA inside of the Altera SoCFPGA Cyclone V chip from Linux user land and using it to accelerate the Map-Reduce computation.

The second problem stems from the fact that the Altera SoCFPGA Cyclone V is an ARM machine, while the Map-Reduce Master is usually an x86 machine. This becomes a problem because the solution to the first problem involves execution of a native helper by the *Worker* on the *Node*.

8.3.1 Utilizing the FPGA

Both the `Map()` and `Reduce()` functions must execute all of the steps presented in Section 7.3 on page 30 every time either of these functions is called. This poses multiple challenges.

The first challenge is the `Map()` function. This function is called for every (key, value) pair of the input data. Therefore, the FPGA would have to be constantly reprogrammed. But the initial measurements point out that it takes 0.5 seconds to reprogram the FPGA, while the computation on the FPGA takes only tens of milliseconds. It is therefore not at all viable to do such a constant reprogramming.

Another challenge is that there can be *Worker*s which require a different computational unit programmed into the FPGA running on the same *Node* in the Map-Reduce cluster. It is therefore necessary to make sure these *Worker*s won't interfere with each other.

Finally, there is a security aspect. The programming of the FPGA and the manipulation with the FPGA bridges can only be performed by the `root` user, while the computation is usually executed as a user task. While security is not a main concern when an arbitrary user process can modify the hardware itself, implementing this kind of a separation is a low hanging fruit and good software design practice.

The proposed solution for all three of the aforementioned challenges is an FPGA multiplexing daemon running on the *Node*. The schematic of operation of the FPGA multiplexing daemon, called FPGAd, which was implemented for the purpose of this text is in Figure 8.7. The full source code for the FPGAd daemon is on the attached CD in the `fpga/src/sw/disco/fpgad` directory. The client code for the FPGAd is in `fpga/src/sw/disco/md5.ext`.

The schematic depicts a single *Node*. There are two distinct Map-Reduce jobs that have their distinct *Mapper*s and *Reducer*s running on this *Node* in parallel. All of those *Mapper*s and *Reducer*s utilize the FPGA to accelerate their computation.

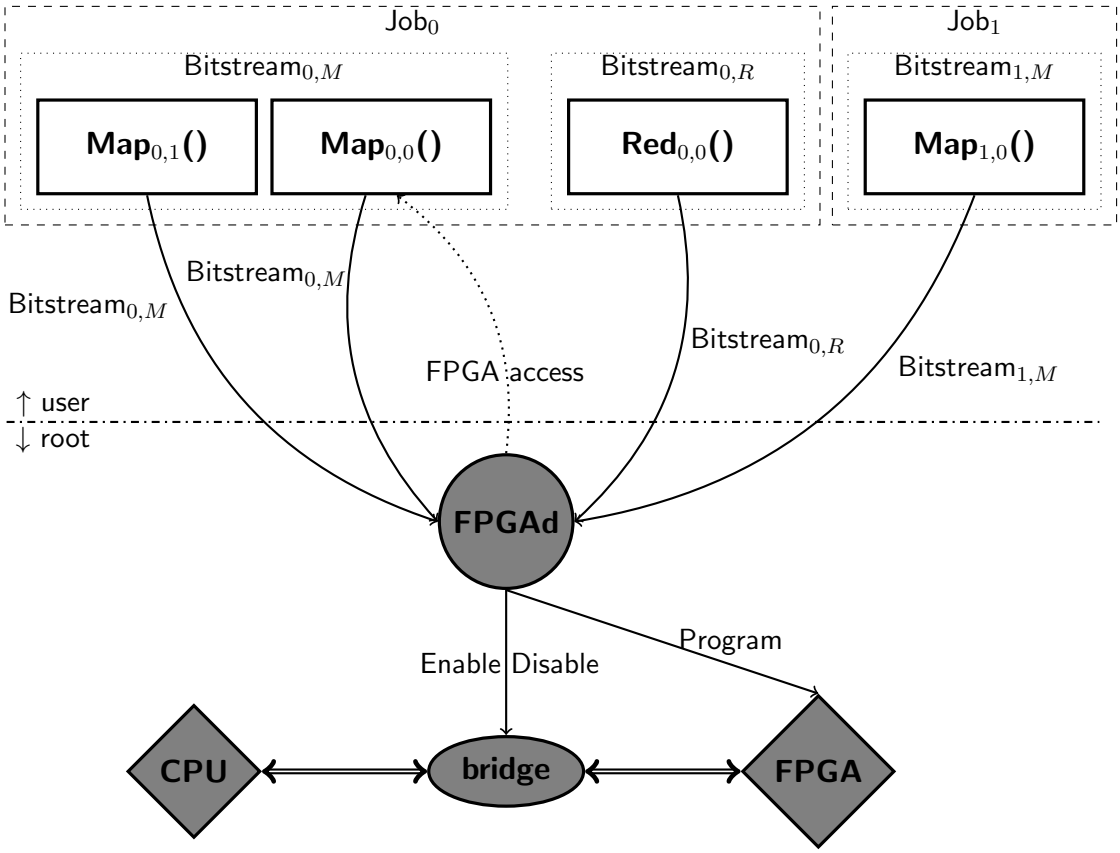


Figure 8.7: FPGA multiplexing using the FPGAd daemon

Notice that there are two *Mapper*s from Job_0 running on the *Node*. Both *Mapper*s of Job_0 share the same FPGA bitstream, in this case the $\texttt{Bitstream}_{0,M}$ to implement the accelerated `Map()` function. The same applies for *Reducer*s and for other Jobs.

To mitigate the first challenge, where the `Map()` function would have to continuously reprogram the FPGA, the `Map()` function provides the path to the FPGA bitstream on the local file system to FPGAd and waits until the FPGAd provides access to the FPGA for the `Map()` function.

Behind the scenes, FPGAd tracks all requests for FPGA access and groups those requests sharing the same FPGA bitstream path. Therefore, the FPGA is only programmed once with a particular bitstream and then the access to the FPGA is granted in sequence to all the `Map()` requests sharing the same FPGA bitstream path.

Note that it is legal to determine if the FPGA bitstream for different `Map()` requests implements the same FPGA contents by comparing the path. Each Disco Job stores the files required for a Job in a single copy on the *Node*. An even more generic solution would be to use a hash of the FPGA bitstream itself, yet this is not implemented.

The second challenge, the synchronization of access to the FPGA, is also solved using FPGAd since all of the `Mapper`s and `Reducer`s must first acquire permission to use the IP block in the FPGA and because FPGAd is in control of re-programming the FPGA itself instead of the *Worker*s.

Finally, FPGAd opens a communication socket through which all the *Work-*

*er*s talk to it. The FPGAd runs as a root user, but the *Worker*s only need access permissions for the FPGAd socket and can thus run as a unprivileged user processes.

8.3.2 Interfacing the FPGA daemon

From the user point of view, once FPGAd grants access to the FPGA, the `Mapper` or `Reducer` is provided with two file descriptors. The provided file descriptors must be `mmap`()ed and the resulting mapping used to access the FPGA.

The first file descriptor provides access to the control registers of the accelerator that is implemented in the FPGA. The other file descriptor is pointing to a reserved area of the RAM. This area can be used by a DMA engine implemented in the FPGA to read data from the RAM. The same area can be used to store the results generated by the FPGA.

Given that the code controlling the FPGA operation requires a lot of control over the execution, in this case, the `Map()` function is implemented in the C language instead of Python.

The Disco framework provides an easy way to call an external program which implements either the `Map()` or `Reduce()` function. This makes it straightforward to implement the FPGA control code in C.

There is a minor catch to the behavior of this interface. Instead of spawning the program implementing the `Map()` or `Reduce()` function for every (key, value) pair in a loop, the program is spawned only once before the `Map()` or `Reduce()` functions are called. The program must read the STDIN for input (key, value) pairs and for every read (key, value) pair must emit one or more (key, value) pairs onto STDOUT.

8.3.3 Native code execution

Compared to the software implementation of the benchmark, where the whole benchmark was implemented in Python, the hardware accelerated benchmark has the `Map()` function implemented in C. The problem is that the Map-Reduce Master node is often an x86 system, while the *Worker* is an ARM system.

The implication is that the program implementing the `Map()` function must be compiled on the *Node* or cross-compiled on the host system. Since the binary uses shared libraries, it is also necessary to make sure that all the *Node*s in the cluster that execute such a program implementing the `Map()` function, have the same system image installed.

In case the cluster contained *Node*s with a different system images, there is a theoretical possibility that the libraries installed on such systems might be missing.

Note that it is not possible to produce a statically linked binary on a contemporary GNU/Linux system with the GNU C Library according to the GNU C Library FAQ [18].

8.3.4 Benchmark amendment

Section 7.2.2 on page 29 pointed out that the FPGA has a setup overhead and therefore benefits from processing data in larger batches. The previous section

showed that the `Map()` function reads a (key, value) pair from STDIN and emits a transformed (key, value) pair to STDOUT.

The easiest way to provide the FPGA with more than a single value during every loop of the `Map()` function is to adjust the input data in the DDFS in such a way that every key has multiple values associated with it. The input chunk reader function for the DDFS to generate such adjusted values is presented in Figure 8.8.

```
def hard_chunk_reader(stream, size, url, params):
    import struct
    ncsums = 2**18
    for j in range(0, 2**10):
        rn = range(ncsums * j, ncsums * (j + 1))
        yield "k", ''.join(struct.pack(">i", i) for i in rn)
```

Figure 8.8: DDFS input data reader implementation, 2^{18} plain texts per key

The `ncsums` variable selects the amount of values associated with one key. In this case, it is set to 2^{18}, being the maximum amount of MD5 blocks that the FPGA can process in one run. It was observed that the DDFS client refuses to load a value which was longer than 1 MiB. To load these adjusted input data into the DDFS, the `-Z 2` option was used to override this software limit. The full command line is presented in Figure 8.9. The data are loaded into the DDFS and identified by the `data:md5:bin256k` *Tag* using the command below, which generates exactly 32 *Blob*s and thus limits the number of *Mapper*s in the cluster to 32.

```
$ PYTHONPATH=$PYTHONPATH:md5/ ddfs chunk -S 4 -Z 2 data:md5:bin256k \
                                  /dev/null                         \
                                  --reader md5.hard_chunk_reader
```

Figure 8.9: DDFS input data reader execution, 2^{18} plain texts per key

A software implementation of the `Map()` function that can work on such data is presented in Figure 8.10. Since the input data for the FPGA implementation differ from the input data for the software benchmark, it is necessary to implement a comparable software benchmark. It would not be possible to draw any conclusion about the performance of the FPGA system compared to the reference x86 system in case the input data were different.

It is easy to notice that this software implementation of the `Map()` function operating on the adjusted data, iterates over the values associated with the single key and does the same operation as `soft_map()` presented in Figure 8.6 for each of the values. This implementation therefore also yields one (key, value) pair for every plain text.

```
def soft_map_multi(kv, params):
    sval = kv[1]
    for j in range(0, len(sval) / 4):
        val = sval[(4 * j):(4 * (j + 1))]
        hsh = hashlib.md5(val).digest()
        yield val[3], val + hsh
```

Figure 8.10: Software Map implementation, 2^{18} plain texts per key

8.4 Benchmark summary

This section presented two distinct variants of the MD5 benchmark. Both of the variants are used in the subsequent chapters to evaluate the hardware. Since this chapter was bulkier, it is a good idea to summarize the key properties of the benchmarks.

To identify the benchmarks more concisely in subsequent chapters, the "n plain texts per 1 key" is henceforth abbreviated to *n PpK*.

The first benchmark variant uses input data in a 1 PpK format, thus resulting in 2^{28} (key, value) pairs of input data. This variant of input data results in the availability of 150 *Mapper*s in the cluster.

The second benchmark variant uses input data in 2^{18} PpK format, thus resulting in 2^{10} (key, value) pairs of input data. This variant of input data results in the availability of 32 *Mapper*s in the cluster.

The number of *Reducer*s available in the cluster is limited by the number of *Shuffle* partitions, which is a configurable property of the Disco Map-Reduce job and is set to 4 by default.

Unless explicitly stated otherwise, every benchmark is always executed 5 times for each configuration.

9. Single node results

Measurements conducted in this chapter are always conducted for a single node. Starting with a single node is important for two orthogonal reasons. One reason is that measuring a single node allows determining the most problematic parts of the computation on the particular type of hardware with the particular Map-Reduce framework. On a single node this will completely bypass the possible overhead introduced by network communications. The other is that measuring single node allows comparing the raw performance of the system in the cluster.

All graphs in this and the following chapters present the power consumption over time of the represented systems. The scale for the consumption is on the left side and the time scale is at the bottom. In order to make it easy to see the relation between the system utilization and the power consumption, the graph also contains the number of running *Mapper*s, *Shuffle* and *Reducer*s in 1 second steps. The scale for these processes is at the right side of the graph.

There are two distinct types of tables in this and the following chapters, one shows the timing statistics for the duration of *Map*, *Shuffle* and *Reduce* stages and the total duration of the Map-Reduce job. The other presents the TpW ratios. All values in the tables are the averages of all conducted tests for a particular configuration unless explicitly stated otherwise and all have a standard deviation value to describe the stability of the data, as explained in Chapter 6.4 on page 26.

9.1 Benchmark results

9.1.1 Reference x86-64 system

1 PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	411 ± 4.24	15 ± 0	195.8 ± 18.14	621.8 ± 16.38
2^{18} PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	317.8 ± 1.64	16 ± 0	187.2 ± 7.16	521 ± 7.71

Table 9.1: Timing: 1x ref. x86 system, 150 Mappers, 4 Reducers

#Partitions	TpW_1	$TpW_{2^{18}}$
4	1717.5 ± 32.95	2109.9 ± 16.31

Table 9.2: TpW: 1x ref. x86 system, 150 Mappers, 4 Reducers

A graph of power consumption of the reference x86 system during the execution of the Map-Reduce benchmark job for 1 PpK input data format is presented in Figure 9.1. A graph for 2^{18} PpK is in Figure 9.2. The duration of Map-Reduce stages are presented in Table 9.1. The resulting average TpW ratios are shown in Table 9.2. the measurement artifacts are on the attached CD in the `measurements/data/x86` directory.

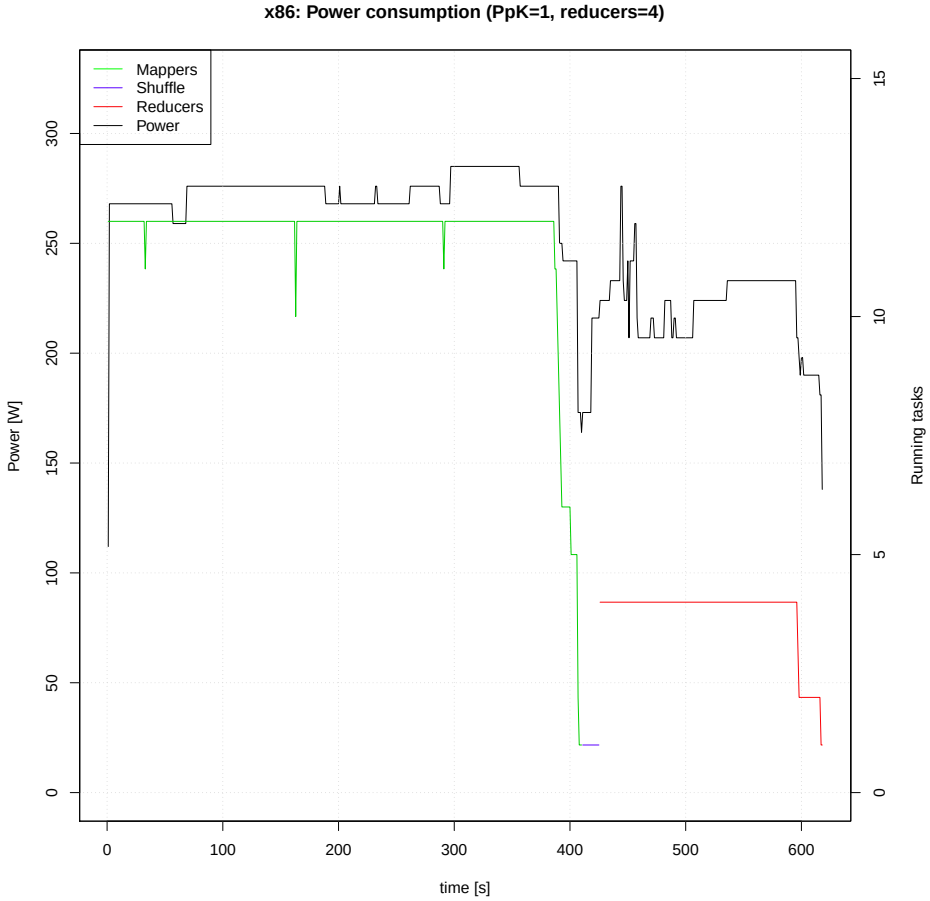


Figure 9.1: Consumption: 1x ref. x86 system, 1 PpK, 150 Mappers, 4 Reducers

Both graphs clearly show that the utilization of the x86 system has a huge impact on the power consumption. It is clearly seen on the left side of the first graph, that the system is fully utilized for roughly the first 400 seconds in case of 1 PpK input data format. The peak power consumption of the system reaches up to 280 W.

In case of the 2^{18} PpK input data format on the second graph, the system shows a drop in the number of *Mapper*s at around 230 seconds and the power consumption starts to fluctuate. This is caused by the 12 logical cores present on the reference x86 system and 32 *Mapper*s available in the cluster. Thus there are not enough *Mapper*s to saturate the system fully for the whole duration of the *Map* stage.

The drop in the system utilization is nonetheless most dramatic in the *Reduce* stage, when only four *Reducer*s are running on the system. This is caused by the number of *Shuffle* partitions, which is 4 by default.

It is noticeable in Table 9.1 that the duration of the Map stage is significantly shorter for the input data 2^{18} PpK format. The performance of the *Map* stage grows because there is less overhead introduced by the Disco framework and the saturation of the CPU cores is thus better when the CPU processes 2^{18} plain texts in a tight loop.

Compared to this, neither of the *Shuffle* or *Reduce* stages are affected in any way by the difference in the format of the input data. The stability in runtime of

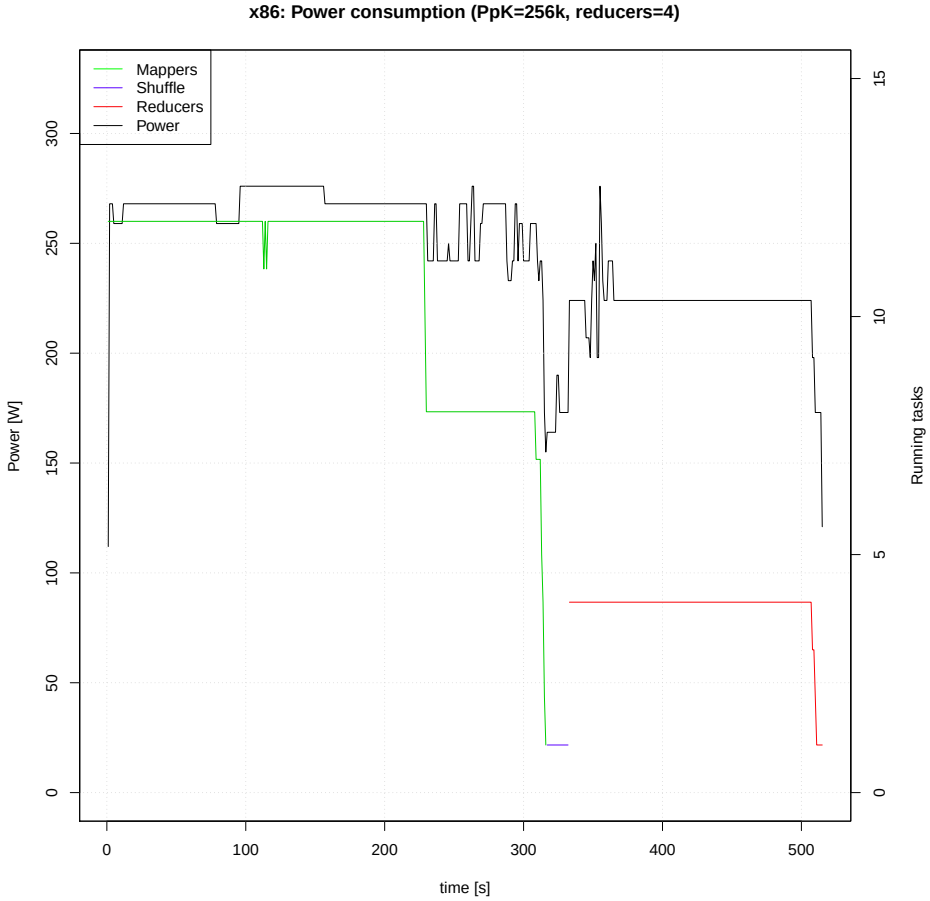


Figure 9.2: Consumption: 1x ref. x86 system, 2^{18} PpK, 32 Mappers, 4 Reducers

both stages is caused by the format of data passed from the *Map* stage into later stages. This intermediate format is the same for both formats of input data as discussed at the end of Section 8.3.4.

9.1.2 ARM Novena system

1 PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	6423.8 ± 31.16	844.2 ± 106.74	1647.6 ± 15.24	8915.6 ± 113.74
2^{18} PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	4665 ± 18.48	800.2 ± 21.09	1643.2 ± 20.36	7108.4 ± 29.92

Table 9.3: Timing: 1x ARM Novena, 150 Mappers, 4 Reducers

A graph of power consumption of the ARM i.MX6 system, the ARM Novena board, during the execution of the Map-Reduce benchmark job for 1 PpK input data format is in Figure 9.3. A graph for 2^{18} PpK is in Figure 9.4. The duration of the Map-Reduce stages are presented in Table 9.3. Table 9.4 shows the resulting

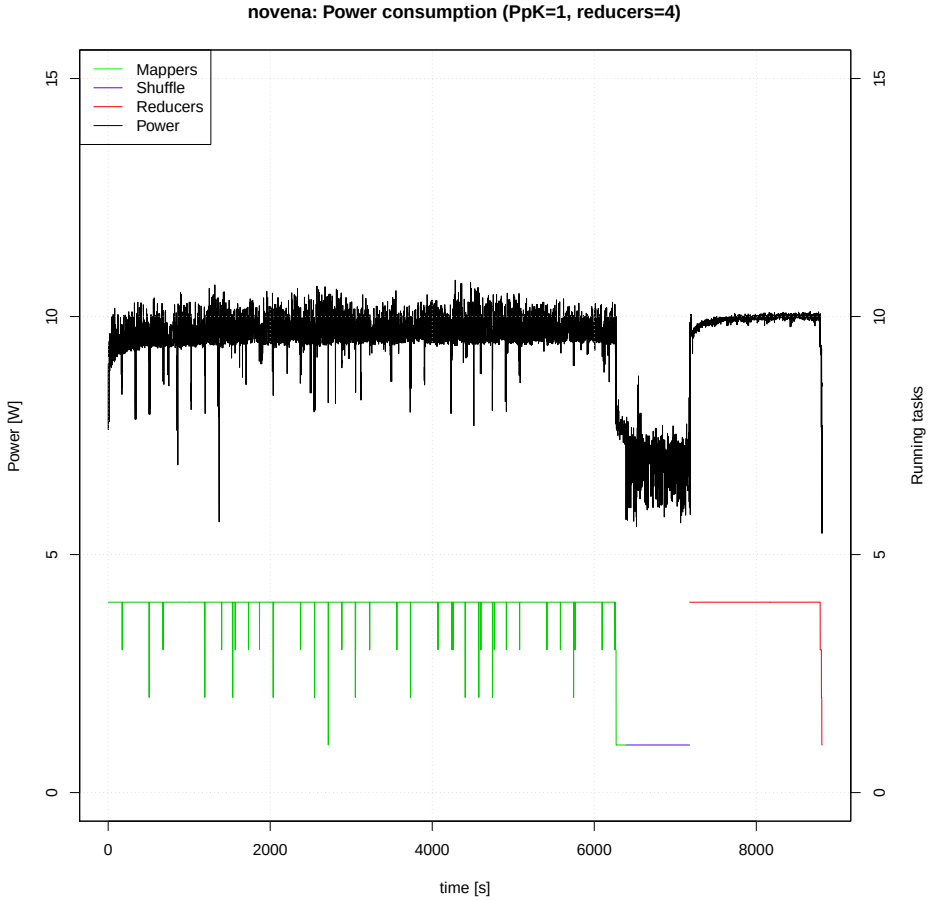


Figure 9.3: Consumption: 1x ARM Novena, 1 PpK, 150 Mappers, 4 Reducers

#Partitions	TpW_1	$TpW_{2^{18}}$
4	3211.64 ± 20.49	3951.75 ± 17.46

Table 9.4: TpW: 1x ARM Novena, 150 Mappers, 4 Reducers

average TpW ratios. All of the measurement artifacts are on the attached CD in the `measurements/data/novena` directory.

Unlike the reference x86 system, the graph clearly shows the ARM Novena system to be fully saturated during both *Map* and *Reduce* stages. This is caused by the lower number of logical CPU cores on the ARM Novena system, which has only four cores. The number of *Mapper*s and *Reducer*s in the cluster is thus large enough to keep the system saturated.

From the Map-Reduce job execution perspective, it is seen that the *Map* stage takes much longer than the *Reduce* stage. This is a noticeable difference compared to the reference x86 system, where both the *Map* and *Reduce* stages take roughly the same amount of time in this particular setup.

A long dip in the power consumption is also seen between the *Map* and *Reduce* stages. This dip is caused by the drop in the system utilization during the *Shuffle* stage. The *Shuffle* stage is running in a single thread only and thus utilizes only one logical core of the machine. Interestingly, the *Shuffle* stage runs proportionally much longer on the ARM Novena system than on the reference x86 system,

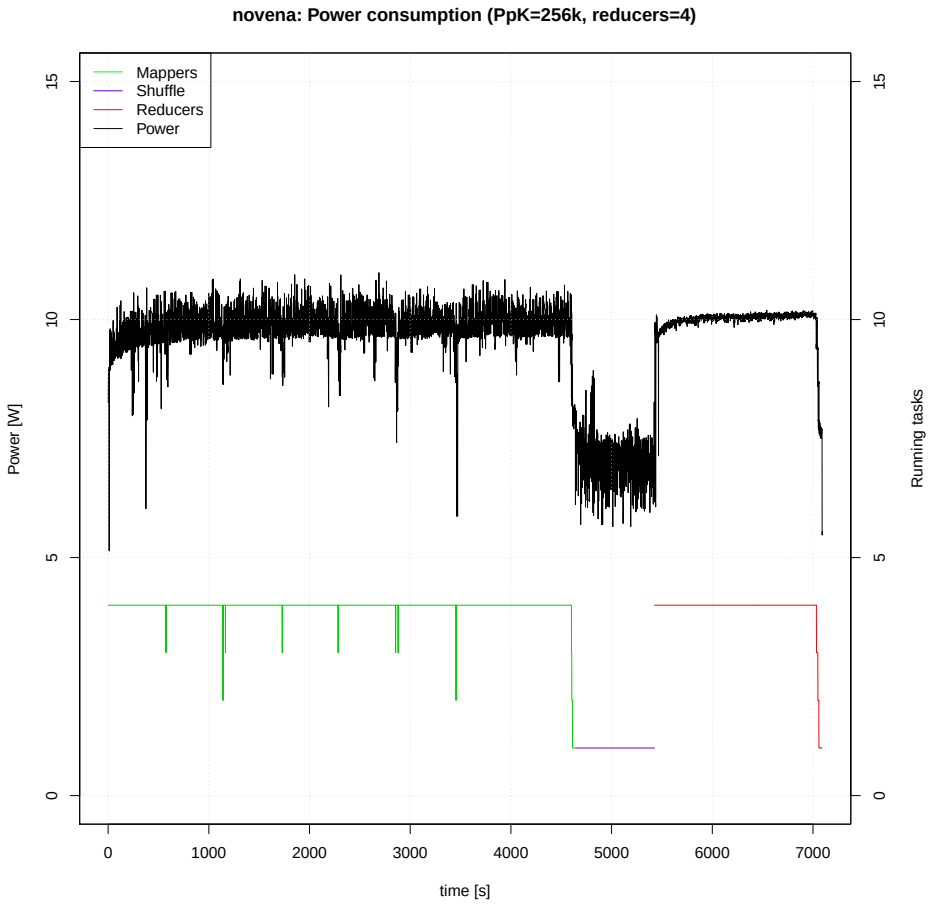


Figure 9.4: Consumption: 1x ARM Novena, 2^{18} PpK, 32 Mappers, 4 Reducers

yet their impact on power consumption due to the under utilization of the system is similar.

Similarly to the reference x86 system, the values in Table 9.3 clarify, that the duration of the Map stage is shorter for the data in the 2^{18} PpK format. It is also clear that neither the *Shuffle* or *Reduce* stages are affected in any way by the difference in the input data format. This confirms the previous reasoning for this behavior presented for the reference x86 system and shows that is also applies to the ARM Novena system.

9.1.3 ARM-FPGA system

A graph of power consumption of the combined ARM-FPGA system, the Altera SoCFPGA system is in Figure 9.5. Compared to the previous two systems, the measurement in this case was conducted only for the 2^{18} PpK input format and was conducted only once. The measurement artifacts are on the attached CD in the `measurements/data/mcvevk-init` directory.

The 1 PpK format of input data was omitted because, as was pointed out in Section 8.3 on page 39, the performance of the computation accelerated by the FPGA is much higher in case the data are supplied in bulk. Analyzing the case where the FPGA is provided with 1 PpK therefore does not yield any gain.

As explained at the end of Section 7.2.2 on page 29, the maximum theoretical

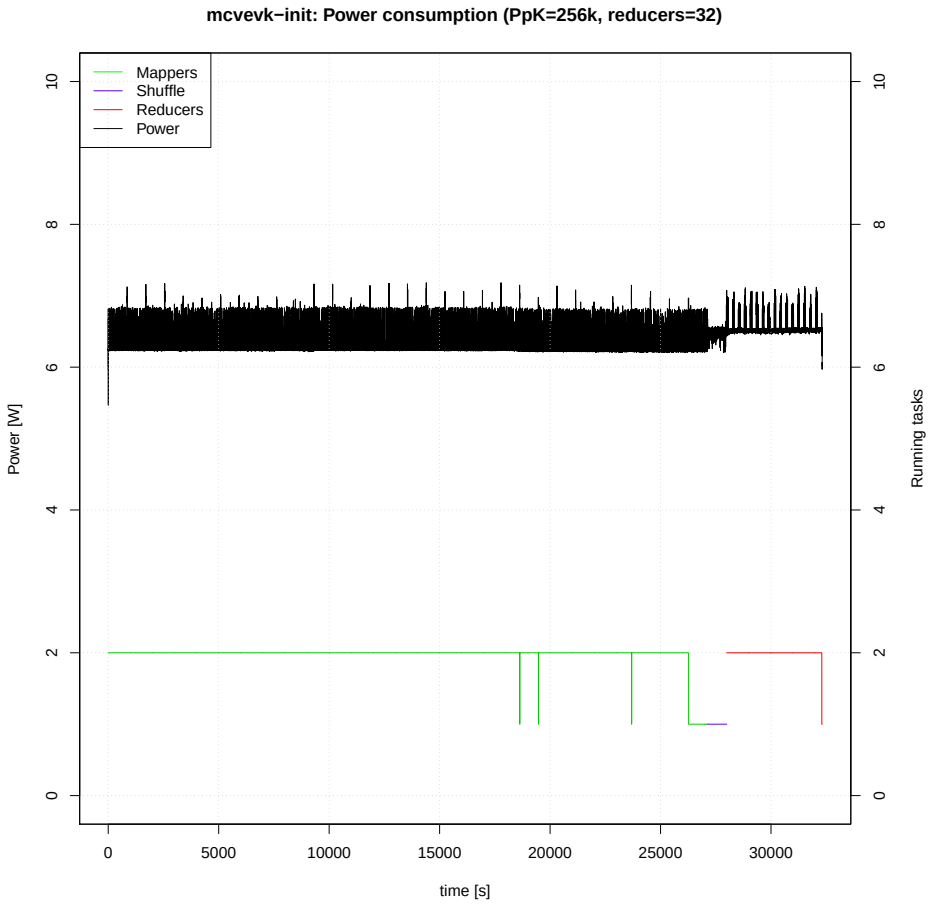


Figure 9.5: Consumption: 1x ARM-FPGA, 2^{18} PpK, 32 Mappers, 4 Reducers

performance of the FPGA is impressively high. Based on these theoretical numbers, the combined ARM-FPGA node is expected to be the fastest of all three observed systems. It is therefore a great surprise that this is not the case and that the combined ARM-FPGA node is in fact the slowest of the three options.

The log messages from the Disco Map-Reduce implementation point out that there is a clear performance bottleneck, doing multiple runs of the same benchmark therefore is not neccessary under these conditions. The result for the ARM-FPGA system in this section thus results from only one single measurement run.

For the purpose of completeness, the TpW ratio of the ARM-FPGA system in this disappointing run of the benchmark is 1306.98, the duration of the *Map* stage is 27117 seconds, shuffle *Shuffle* stage is 875 seconds, the *Reduce* stage is 4325 seconds and the total duration is 32317 seconds.

9.2 Further questions

These initial measurements conducted on a single node give a good idea about which part of the Map-Reduce computation and on which platform needs to be inspected further. The questions listed below come from analyzing the results.

9.2.1 Number of partitions

The number of *Mapper*s and *Reducer*s in the cluster plays an interesting role in the saturation of the whole cluster. In case either number is low, the cluster ends up being under saturated and some computational units are therefore left idle. This severly impacts the TpW ratio.

By observing the results from all three platforms above, it is easily noticed that the reference x86 system is heavily under saturated in the *Reduce* stage, while both of the ARM systems are not. Since the reference x86 system has 12 logical cores and the ARM systems have four in case of the ARM Novena and two in case of the ARM-FPGA respectively, it is immediately clear why the ARM systems are fully saturated when there are four *Reducer*s running in the cluster. The reference x86 system, on the other hand, could host up to 12 *Reducer*s and is thus under saturated.

The default configuration of the Disco Map-Reduce uses four Shuffle partitions, into which the *Shuffle* stage divides the results from the *Map* stage, resp. from the *Combiner*. The number of partitions imposes an upper bound on the number of *Reducer*s to which the particular node exports *Shard*s.

Since both the saturation of the system and the duration of the stages of the Map-Reduce job have an impact on the TpW ratio, it is of interest to determine how the TpW ratio changes for different numbers of Shuffle partitions. This is researched in Chapter 10 on page 52.

9.2.2 ARM Cluster performance

While it is already obvious from this chapter that the ARM Novena system is fully saturated in both the *Map* and *Reduce* stages, the measurement conducted in this chapter was conducted only on a single node.

The obvious benefit of learning about the ARM system in the cluster setting is that it is possible to determine how much the performance of the cluster of ARM systems will drop with the increasing number of Nodes.

It is also possible to compare the impact of the *Shuffle* stage onto the TpW ratio of the ARM cluster. In the cluster setting, the data are actually exchanged between the *Mapper*s and *Reducer*s over the network.

By learning about the behavior of the Shuffle stage in a cluster setting, it should be possible to determine if it is viable to use lots of ARM systems to reach the same duration of the Map-Reduce job computation with a lower TpW ratio.

This cluster impact on the ARM system is researched in Chapter 11 on page 59.

9.2.3 ARM-FPGA system performance

The initial measurement of the efficiency of the ARM-FPGA system is clearly a disappointment. Not only does the combined ARM-FPGA system show very poor computational performance. Also the TpW ratio of the combined ARM-FPGA system is even worse than the one of the reference x86 system.

It is very important to learn why the combined ARM-FPGA system suffers from such a performance loss, where the performance is lost and how the situation

can be improved. The performance bottleneck of the ARM-FPGA system is researched in Chapter 12 on page 63.

10. Partitioning impact

The number of Shuffle partitions imposes an upper bound on the number of Reducers to which the Nodes in the Disco cluster export Shards and thus also imposes an upper bound on the number of Reducers in the cluster. The measurements in this chapter are still conducted on a single Node to provide results comparable to the initial measurements.

Since the reference x86 system was not fully saturated due to the insufficient amount of Reducers in the Disco cluster, it is of interest to learn about the impact of the number of partitions on the TpW ratio of the systems. It is also interesting to learn about the impact in case of the ARM Novena system.

The measurement was not conducted on the combined ARM-FPGA system. The combined ARM-FPGA system implements the same type of ARM cores as the ARM Novena system, yet there are only two such cores and running at a lower clock speed. This test would thus not yield any new information compared to running the test on the ARM Novena machine.

10.1 Impact on x86

1 PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	411 ± 4.24	15 ± 0	195.8 ± 18.14	621.8 ± 16.38
8	409.4 ± 2.07	19 ± 0	126.2 ± 2.05	554.6 ± 2.79
16	410 ± 3	25.2 ± 0.45	126.8 ± 1.48	562 ± 2.92
24	412.8 ± 2.77	22 ± 0	132.4 ± 3.36	567.2 ± 5.59
32	414.4 ± 2.7	36 ± 0	114.6 ± 0.89	565 ± 2.92
2^{18} PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	317.8 ± 1.64	16 ± 0	187.2 ± 7.16	521 ± 7.71
8	319.2 ± 1.3	17 ± 0	129.6 ± 1.82	465.8 ± 2.77
16	320.4 ± 1.14	16 ± 0	133 ± 6.75	469.4 ± 7.23
24	319.8 ± 1.64	17 ± 0	129 ± 1.58	465.8 ± 2.59
32	319.4 ± 3.44	17 ± 0	116.2 ± 1.64	452.6 ± 2.07

Table 10.1: Timing: 1x ref. x86 system, 32/150 Mappers, 4/8/16/24/32 Reducers

#Partitions	TpW_1	$TpW_{2^{18}}$
4	1717.5 ± 32.95	2109.9 ± 16.31
8	1855.85 ± 9.46	2233.96 ± 8.73
16	1837.48 ± 6.05	2220.78 ± 24.81
24	1827.72 ± 15.37	2229.98 ± 10.01
32	1826.4 ± 5.34	2263.85 ± 9.95

Table 10.2: TpW: 1x ref. x86 system, 32/150 Mappers, 4/8/16/24/32 Reducers

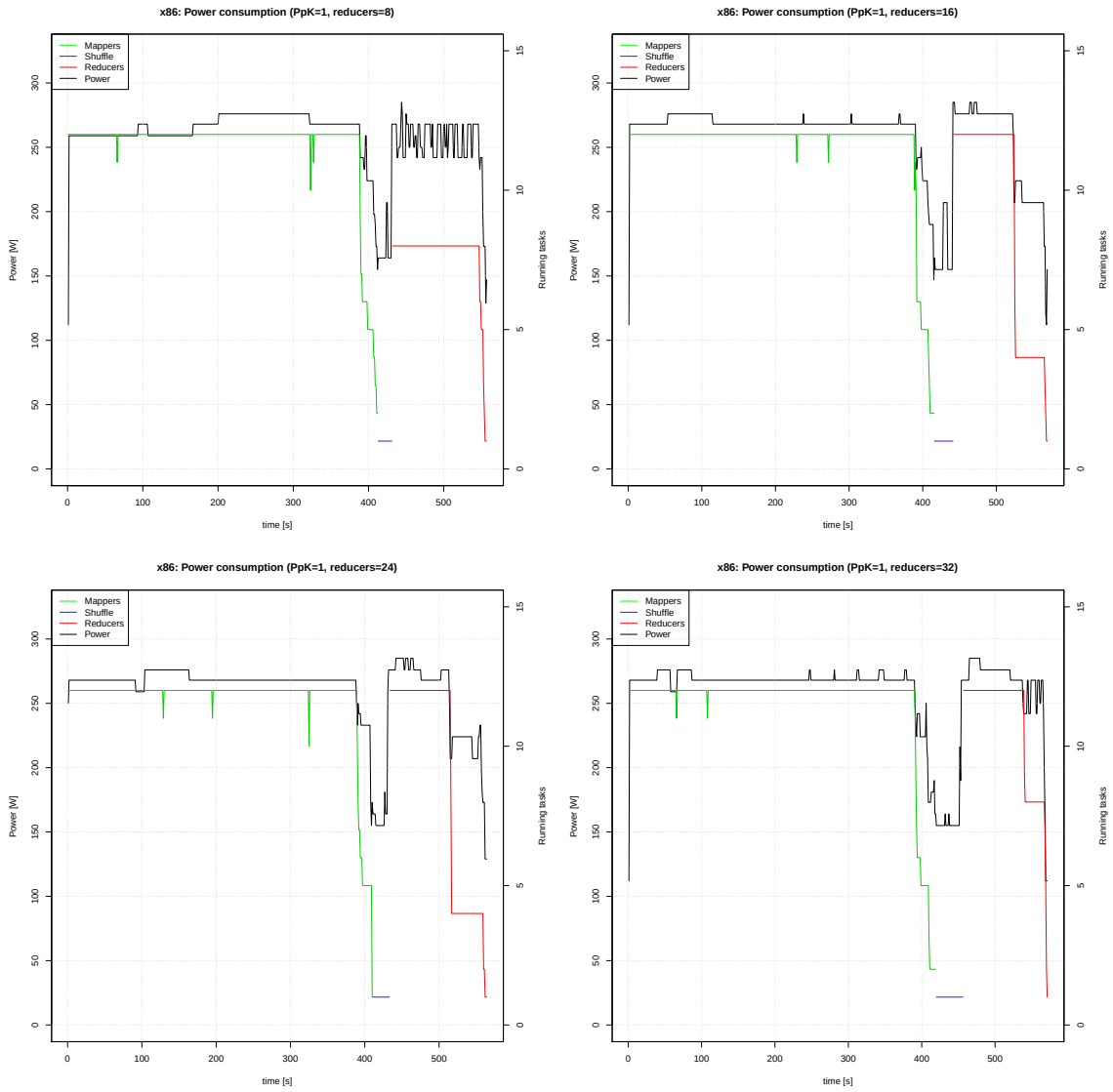


Figure 10.1: Consumption: 1x ref. x86 system, 1 PpK, 150 Mappers, 8/16/24/32 Reducers

The results of the automated testing on the reference x86 system are presented in Table 10.1 and Table 10.2, which show the timing of distinct stages of the Map-Reduce job and a resulting TpW for distinct number of Shuffle partitions. A graphical representation of these results is presented in Figure 10.1 and Figure 10.2 for input data in the format of 1 PpK and 2^{18} PpK respectively. All of the measurement artifacts are on the attached CD in the `measurements/data/x86` directory.

By examining the timings of the distinct stages of the Map-Reduce algorithm in Table 10.1, it is clearly seen that the Map stage benefits greatly from using the input data in format of 2^{18} PpK, yet the Map stage timing does not change with increasing number of Shuffle partitions.

The impact of input data format on the performance of Map stage is explained in Section 9.1.1 on page 44 by the decreased overhead of the framework in case the data are processed in a tighter loop. The lack of change in the timing of the Map function in proportion to the number of Shuffle partitions is expected because

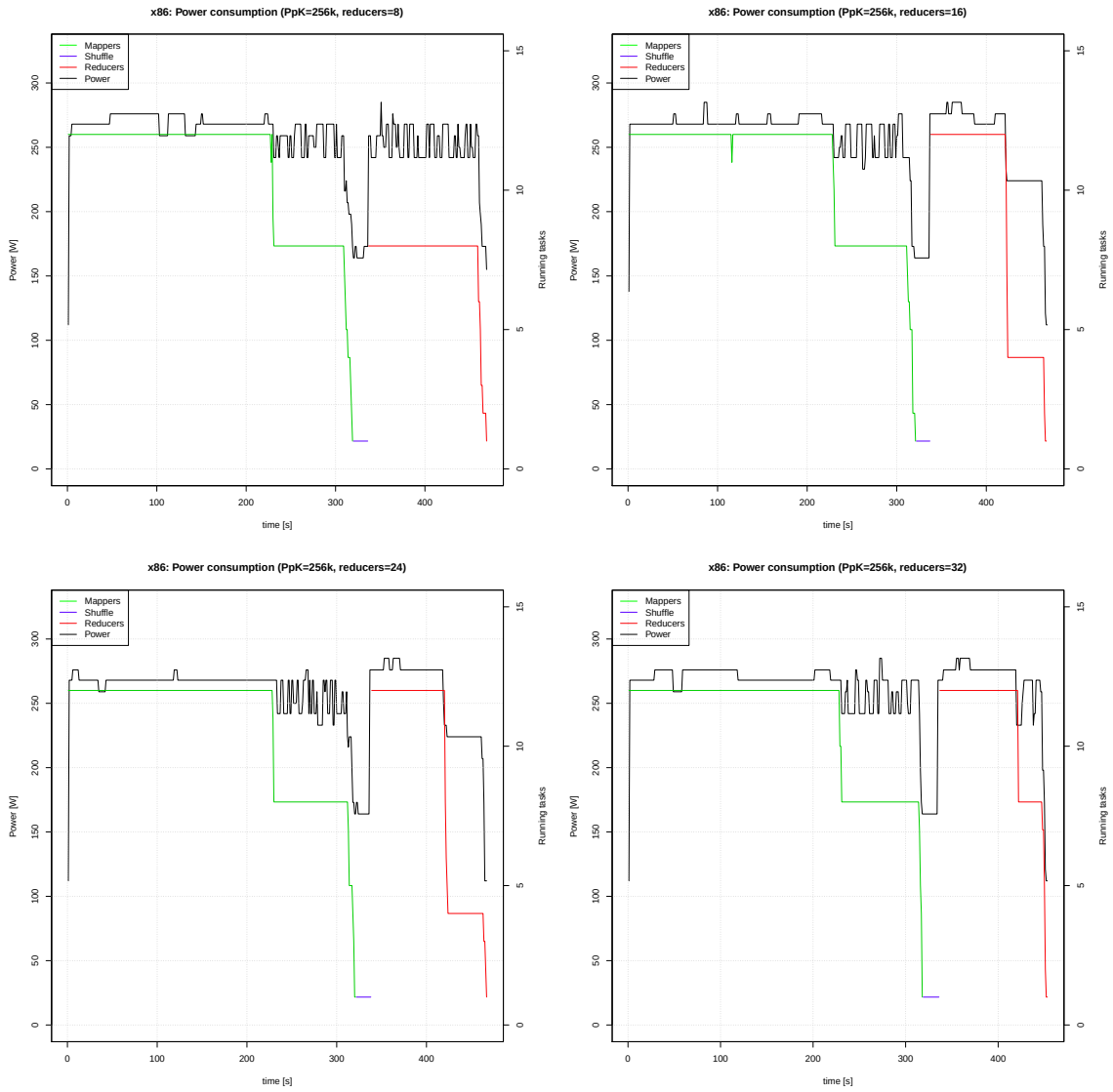


Figure 10.2: Consumption: 1x ref. x86 system, 2^{18} PpK, 32 Mappers, 8/16/24/32 Reducers

the Map stage does not take the number of Shuffle partitions into consideration in any way.

The behavior of the Shuffle stage is also not affected by the changing number of Shuffle partitions. Since the Map function generates the same amount of intermediate data in case of both types of input data formats, it is expected that the Shuffle function is not affected by the change of input data format. The overhead of the partitioning function used in the Shuffle stage is apparently not affected by the number of partitions which it generates.

Finally, the Shuffle stage uses the same data for both input data formats and thus outputs the same data as well, it is thus expected that the timing of the Reduce stage does not differ for both input data formats. On the other hand, it is clearly seen that the duration of the Reduce stage is decreasing with the growing number of Shuffle partitions. This is caused by the growing saturation of the system because there are more Reducers available.

The growing number of Reducers also positively influences the TpW ratio of

the system, which peaks around value 2250 in case of the input data in format of 2^{18} PpK. On the other hand, the TpW ratio of the reference x86 system is not changing in any significant way starting from 8 shuffle partitions, which means that for this particular test case, the TpW ratio is close to it's upper bound.

10.2 Impact on ARM

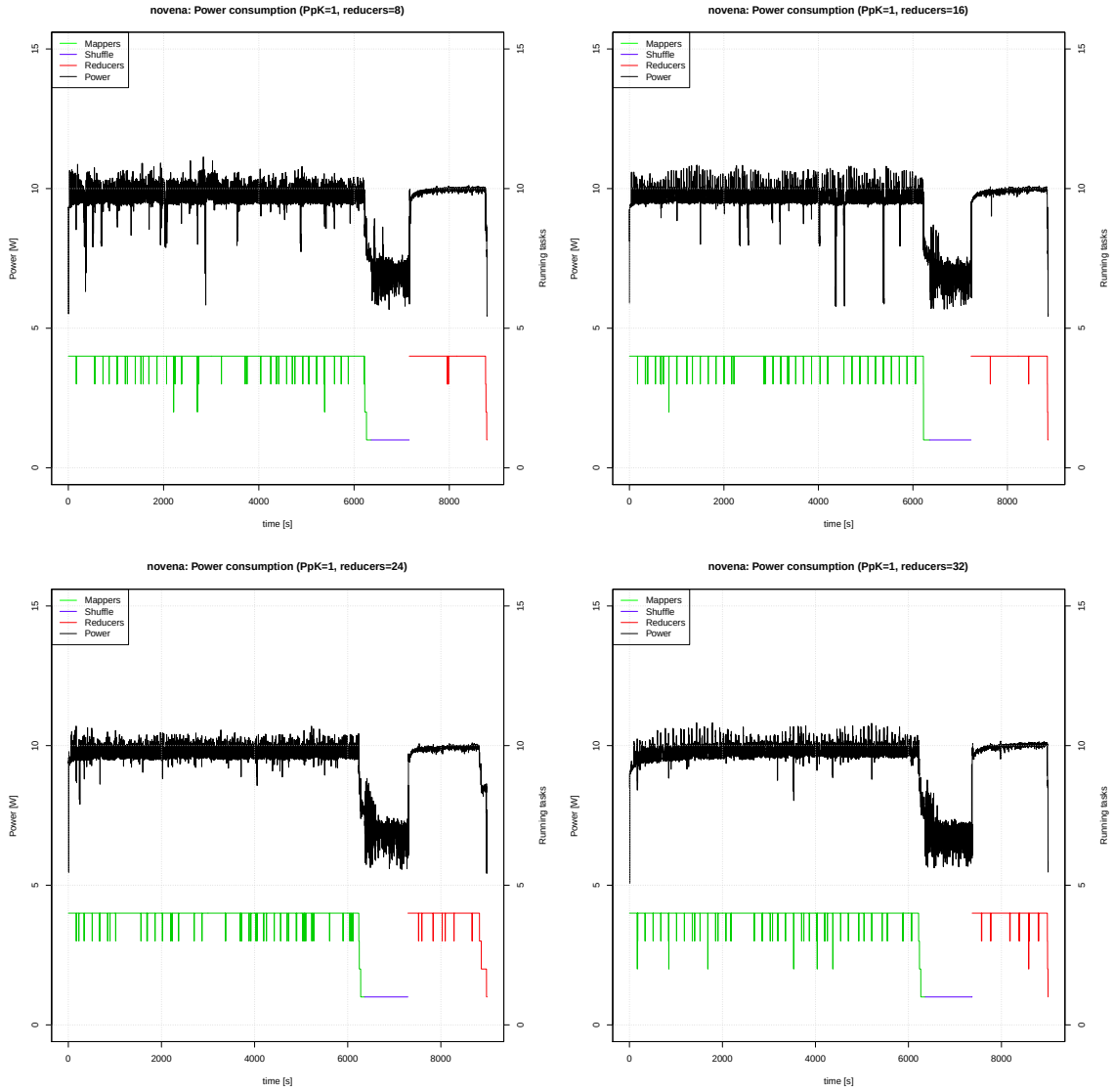


Figure 10.3: Consumption: 1x ARM Novena, 1 PpK, 150 Mappers, 8/16/24/32 Reducers

The results of the automated testing on the ARM Novena system are presented in Table 10.3 and Table 10.4, which show the timing of distinct stages of the Map-Reduce job and a resulting TpW for distinct number of Shuffle partitions. A graphical representation of these results is presented in Figure 10.3 and Figure 10.4 for input data in the format of 1 PpK and 2^{18} PpK respectively. All of the measurement artifacts are on the attached CD in the `measurements/data/novena` directory.

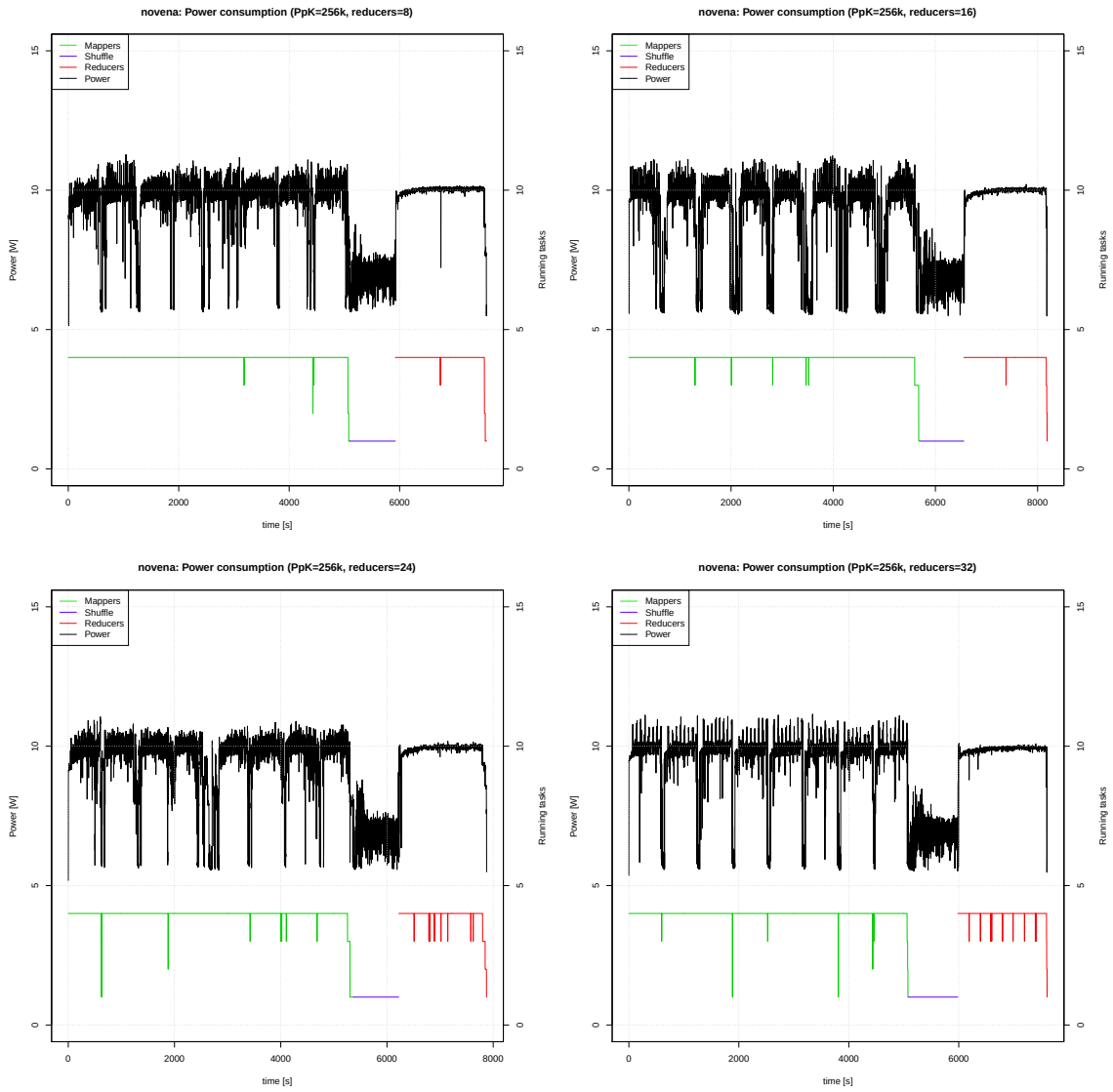


Figure 10.4: Consumption: 1x ARM Novena, 2^{18} PpK, 32 Mappers, 8/16/24/32 Reducers

By examining the timings of the distinct stages of the Map-Reduce algorithm in Table 10.3, just like in case of the reference x86 system, using the input data in format of 2^{18} PpK results in a shorter Map stage. It is noticeable that in case of the 1 PpK format of input data, the duration of the Map stage is stable across the changing number of Shuffle partitions, which is aligned with the behavior of the reference x86 system.

On the other hand, in case of the 2^{18} PpK input data format, the duration of the Map stage fluctuates much more. This is seen in particular for 4 and 16 reducers, which are the peak values. Yet, the mean value is still 5186.04 seconds, which is aligned with the remaining measurements for 8, 24 and 32 Shuffle partitions.

Compared to the reference x86 system, the behavior of the Shuffle stage is slightly affected by the changing number of Shuffle partitions for both input data formats. Apparently, the overhead of the partitioning is much higher on ARM.

By analyzing the Disco code, in particular the default partitioning function for

1 PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	6423.8 ± 31.16	844.2 ± 106.74	1647.6 ± 15.24	8915.6 ± 113.74
8	6364.2 ± 10.43	799.4 ± 19.58	1634.8 ± 4.92	8798.4 ± 28.48
16	6322.4 ± 31.11	850 ± 14.92	1631.8 ± 7.12	8804.2 ± 38.75
24	6357.8 ± 12.38	908.4 ± 19.17	1681.6 ± 19.96	8947.8 ± 28.5
32	6378.2 ± 17.14	1008 ± 32.95	1630.4 ± 4.04	9016.6 ± 36.05
2^{18} PpK				
#Partitions	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
4	4665 ± 18.48	800.2 ± 21.09	1643.2 ± 20.36	7108.4 ± 29.92
8	5175.2 ± 96.93	816.6 ± 16.13	1649.4 ± 8.91	7641.2 ± 97.15
16	5675 ± 125.4	866.2 ± 53.09	1649.8 ± 21.51	8191 ± 146.83
24	5199.2 ± 161.81	855.4 ± 14.74	1670 ± 19.74	7724.6 ± 150.98
32	5215.8 ± 100.61	905 ± 25.76	1625.6 ± 16.1	7746.4 ± 107.23

Table 10.3: Timing: 1x ARM Novena, 32/150 Mappers, 4/8/16/24/32 Reducers

#Partitions	TpW_1	$TpW_{2^{18}}$
4	3211.64 ± 20.49	3951.75 ± 17.46
8	3222.77 ± 8.49	3783.25 ± 30.41
16	3233.09 ± 14.44	3600.06 ± 24.36
24	3175.26 ± 21.28	3691.47 ± 48.21
32	3143.68 ± 7.36	3721.68 ± 31.32

Table 10.4: TpW: 1x ARM Novena, 32/150 Mappers, 4/8/16/24/32 Reducers

a Map-Reduce job, implemented by the `default_partition()` method, in the file `lib/disco/worker/classic/func.py`, it is seen that the partitioning function is an MD5 hash of the *key* modulo the number of Shuffle partitions. This implies that the only change in the partitioning function itself for different number of Shuffle partitions is a calculation of modulo with different divisor. It thus turns out that the computation of the partitioning function itself is not responsible for the slowdown.

Instead, it is worth analyzing the `map_shuffle()` method itself, which is defined in `lib/disco/worker/classic/worker.py`. This method does a lot of I/O operations and the number of file descriptors is growing with the number of Shuffle partitions. The Python suffers from a severe slowdown when doing I/O operations on ARM, which is further discussed in Chapter 12 on page 63.

Similarly to the reference x86 system, the Shuffle stage generates the same data for both formats of input data. It is thus not surprising that the Reduce stage is not influenced by the format of input data.

It is clearly seen that the system is fully saturated in all cases during the Reduce stage as well. It is therefore not very surprising that the Reduce stage takes approximately the same time for both formats of input data and for a changing number of Shuffle partitions.

The input data in format of 2^{18} PpK exposes a very interesting behavior, which is clearly seen in Figure 10.4. It is noticeable that independently of the number of Shuffle partitions, each graph shows exactly eight dips during the Map

stage, where the last dip fluently transitions into the Intermediate stage. This behavior is caused by all four CPU cores finishing a Map approximately at the same time, which triggers a bulk write into the local storage.

Subsection 8.3.4 on page 41 explains that there are 32 Mappers running in the cluster in the test with input data in format of 2^{18} PpK. This means there are $\frac{2^{28}}{2^{18}} = 2^{10}$ (key, value) pairs available in the cluster, which in turn implies that each Mapper has $\frac{2^{10}}{32} = 32$ (key, value) pairs associated with it. Each (key, value) pair contains 2^{18} PpK, which upon processing as explained in Section 5.2 on page 21 result in 20 Byte long entry each. This means that each Mapper produces over $32 * 2^{18} * 20B = 160MiB$ worth of data upon it's completion. Since the Mapper produces (key, value) pairs and the 160 MiB of data are only the value part, there is some more overhead for the key part even.

The local storage used on the ARM Novena system is an SDXC card and the write throughput of the SDXC card is 13 MiB/s tops[1] . Since four Mappers finish at the same time, they produce an immediate demand to store $4 * 160MiB = 640MiB$ of data onto a storage which can do up to 13 MiB/s. Because the Mapper waits until the data are actually written to the disk before it exits and because the CPU core is not saturated during this stage, the power consumption of the system drops, thus creating the dip which is seen in the graph. And since there are 32 Mappers in the system and four CPU cores, this particular situation happens 8 times in total, which results in the aforementioned 8 dips in the graph.

10.3 Efficiency comparison

By comparing the TpW ratio of the reference x86 system in Table 10.2 and of the ARM Novena system in Table 10.4, it is crystal clear that for all tested amounts of Shuffle partitions, the ARM Novena system is significantly more power efficient than the reference x86 system in this benchmark in a single-node configuration.

It is nonetheless noticable that the duration of the computation on the ARM Novena system is much longer compared to the reference x86 system. It is therefore of great interest to learn about the behavior of the ARM Novena system in a cluster setting, to learn how does such an ARM system scale.

[1]The write throughput was tested repeatedly using the `dd(1)` command, `dd if=/dev/zero of=/tmp/test.bin bs=5M count=32` and fluctuates between 11.9 MiB/s and 14.1 MiB/s for such bulk writes.

11. ARM Cluster benchmark

So far, all the tests were conducted on a single system. This chapter finally investigates the behavior of multiple ARM systems in a Map-Reduce cluster setting. The obvious reason is to learn about the possible surprises which might affect the TpW rating of such a cluster.

The test in this chapter was conducted for a cluster consisting of 2, 3 and 4 nodes. In case of the 2-node configuration, the cluster was assembled from two identical ARM Novena systems. Since these systems are difficult to obtain, the third and fourth node in the cluster was supplanted by an comparable Freescale i.MX6Quad-based prototype board with the same configuration as the ARM Novena system. Using such system assures that the timings of the 3-node and 4-node test are comparable to a cluster of 3 or 4 ARM Novena systems.

Since both the ARM Novena system and reference x86 system show better TpW ratio when using input data in the format of 2^{18} PpK, the test in this chapter is not conducted for the input data in format of 1 PpK.

The benchmark was conducted only for a configuration with 32 Shuffle partitions. This is necessary, since in a 4-node configuration, each node is assigned up to 8 partitions. Since each node has 4 CPU cores, using a lower number of Shuffle partitions would pose a risk of triggering under-saturation of the system.

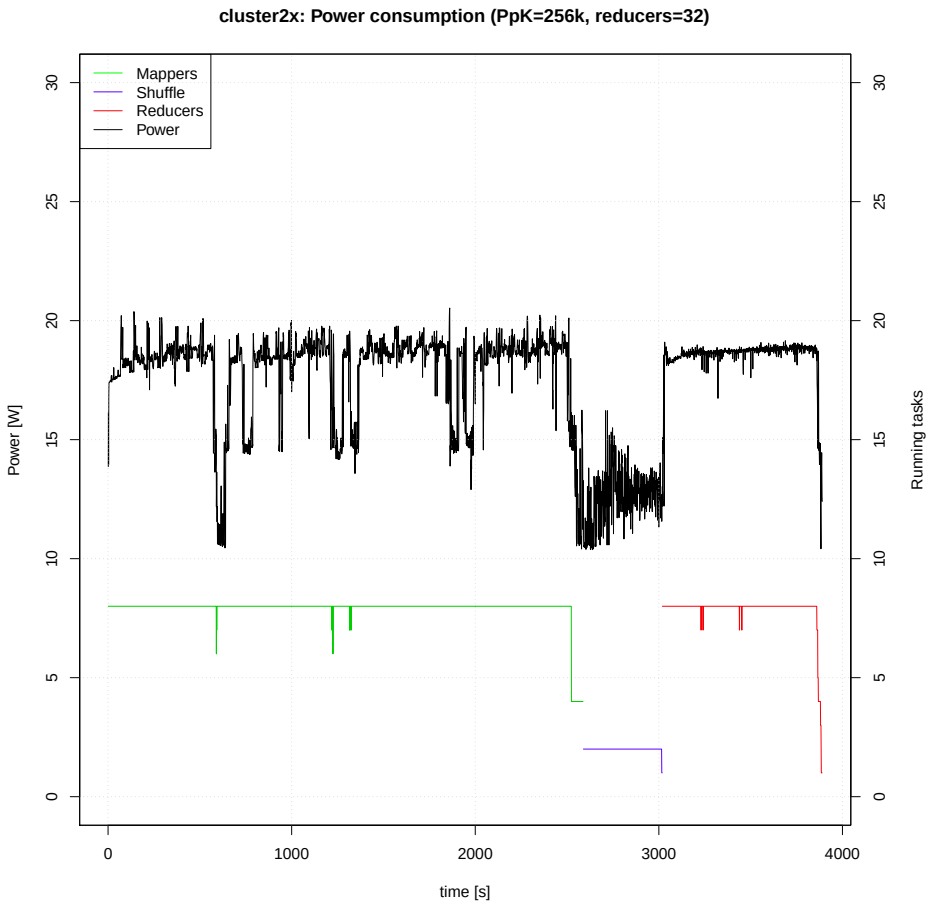


Figure 11.1: Consumption: 2x ARM Novena, 2^{18} PpK, 32 Mappers, 32 Reducers

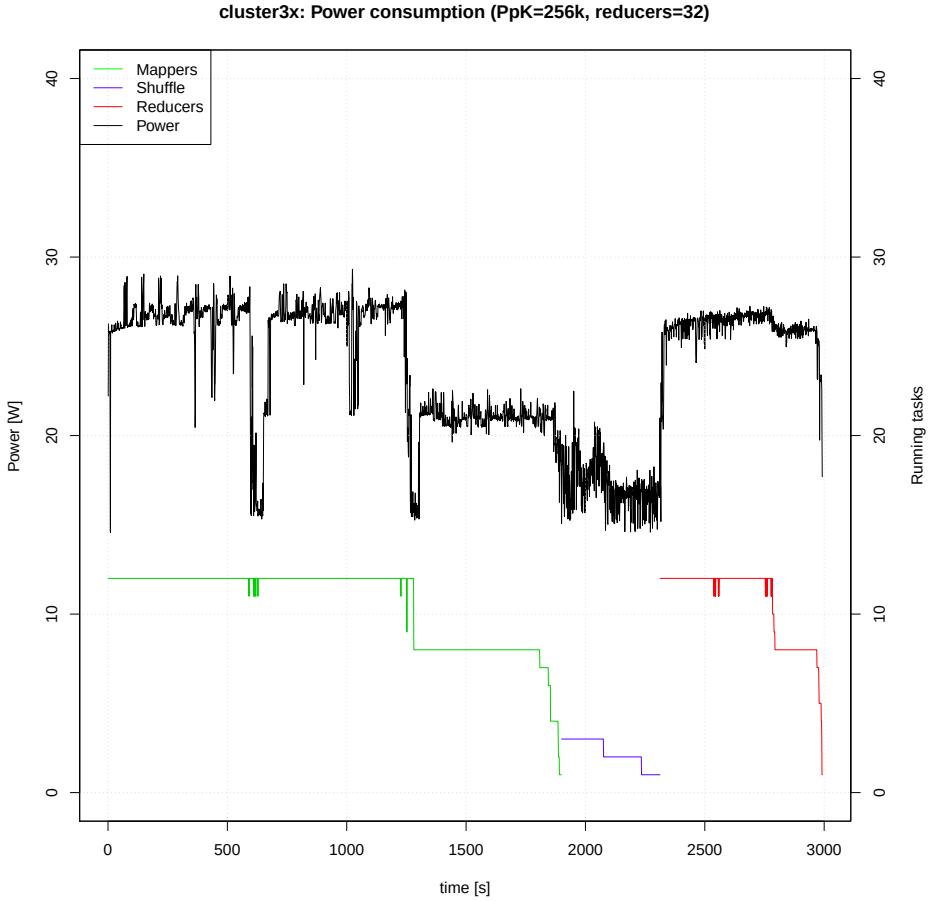


Figure 11.2: Consumption: 3x ARM Novena, 2^{18} PpK, 32 Mappers, 32 Reducers

2^{18} PpK				
#Nodes	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
1	5215.8 ± 124.93	905 ± 31.98	1625.6 ± 19.99	7746.4 ± 133.14
2	2545.6 ± 31.33	465.8 ± 49.29	868.6 ± 3.58	3880 ± 36.49
3	1886.6 ± 23.24	378.8 ± 59.51	671 ± 6.33	2936.4 ± 69.4
4	1283.2 ± 14.46	276.4 ± 32.58	473.2 ± 5.72	2032.8 ± 35.05
k for 2^{18} PpK				
#Nodes	$k_{t_{map}}$	$k_{t_{shuffle}}$	$k_{t_{reduce}}$	$k_{t_{total}}$
1	1.000	1.000	1.000	1.000
2	0.488	0.515	0.534	0.501
3	0.361	0.419	0.413	0.376
4	0.246	0.305	0.291	0.262

Table 11.1: Timing: 1x-4x ARM Novena cluster, 32 Mappers, 32 Reducers

The results of the automated testing on the ARM cluster are presented in Table 11.1 and Table 11.2, which show the timing of distinct stages of the Map-Reduce job and a resulting TpW of the entire cluster for a distinct number of cluster nodes. The tables also presents coefficients k_{value} for each value, which represent the difference between that value for one Node configuration and the particular n-Node configuration. All of the measurement artifacts are on the at-

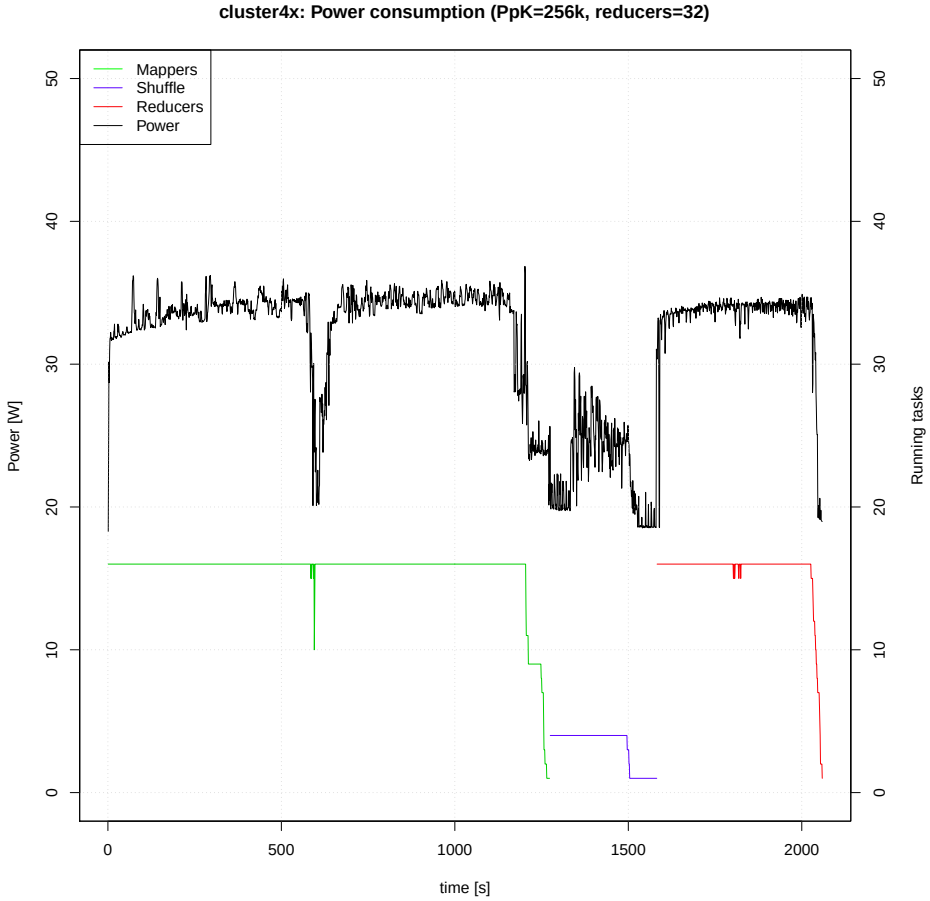


Figure 11.3: Consumption: 4x ARM Novena, 2^{18} PpK, 32 Mappers, 32 Reducers

#Nodes	$TpW_{2^{18}}$	$k_{TpW_{2^{18}}}$
1	3720.61 ± 38.81	1.000
2	3931.19 ± 24.62	1.057
3	3731.98 ± 80.01	1.003
4	4146.21 ± 51.03	1.114

Table 11.2: TpW: 1x-4x ARM Novena cluster, 32 Mappers, 32 Reducers

tached CD in the `measurements/data/novena`, `measurements/data/cluster2x`, `measurements/data/cluster3x` and `measurements/data/cluster4x` directories for 1, 2, 3 and 4 node configurations respectively.

A graphical representation of these results is presented in Figure 11.1, Figure 11.2 and Figure 11.3 for 2, 3 and 4 node cluster respectively.

A particularly low-hanging fruit is seen by comparing the graphs for each cluster configuration and observing the dips in power consumption during the Map stage. This behavior was explained for a single Node at the end of Section 10.2 on page 55. In the cluster case, it is noticeable that for two Nodes, there are four dips in total, for three nodes, there are three dips and for four nodes, there are two dips. This is caused by the decreasing number of Mappers running on each of the nodes and thus also a decreasing number of the bulk write events happening on each node.

Another observation easily derived from the graphs is that in case of a three node cluster configuration, the cluster shows a dip in number of Mappers running in the cluster just before it enters the Intermediate stage. This is caused by the 32 Mappers available in the cluster, which are assigned to the three 4-core machines in the cluster. Clearly, at some point, some cores must become idle because there are no Mappers for them anymore. The cores becoming idle even shows in a decreased TpW rating, as is seen in Table 11.2.

It is now established that the test using three nodes is obscured by the under-saturation of the systems in the cluster, thus this test is not considered in the further paragraphs.

The timings of each stage of the Map-Reduce job show an interesting behavior. The total duration of the test is slightly increasing with increasing number of Nodes, as seen in Table 11.1 in the $k_{t_{total}}$ column. The same slight increase in duration is noticeable for both Shuffle and Reduce stage. On the contrary, the Map stage shows a decrease in the it's duration, which is mostly caused by the decreasing number of the bulk write events.

The resulting TpW rating is also affected by the changes in the timing and is in fact showing a slightly growing trend with the growing number of nodes in the cluster. This slightly growing trend is very important, since it proves that such an ARM cluster does not suffer from severe performance degradation with a growing number of nodes.

It would be of high interest to learn how does the TpW ratio and timings change with even higher number of nodes than those that were used in this chapter. It is seen that the ARM cluster efficiency increases with the growing number of Nodes, yet it is not clear when, and if ever, does this increasing trend end. This is clearly an promising task for future research.

12. ARM-FPGA performance

The initial measurement of the ARM-FPGA system points out that there is an apparent problem when using the ARM-FPGA system in the Map-Reduce cluster. For reason yet unknown, the performance of the ARM-FPGA system in the Map-Reduce cluster is very poor.

The process in which the data are passed from the Disco Map-Reduce framework into the FPGA and back is shown on Figure 12.1 It is clearly seen that the process can be separated into two parts.

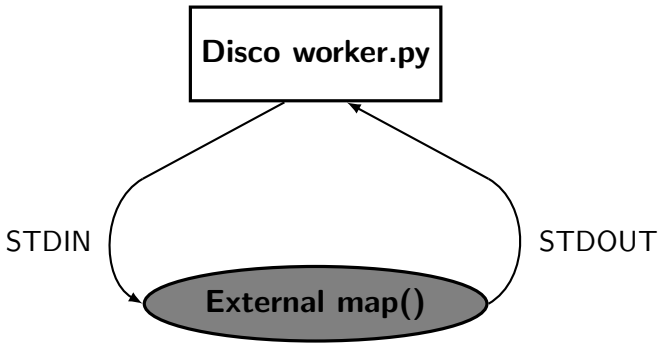


Figure 12.1: Disco interface to external Map implementation

The first part is executed in the context of the Disco framework, where the data are formatted and passed onto the STDIN of the external implementation of the Map() function and retrieved from the STDOUT. The second part is the external Map() function itself, which again re-formats the data and passes them onto the FPGA and collects the results from the FPGA.

Both parts must be analyzed to determine where is the performance loss coming from.

12.1 External Map performance

The performance of the external Map implementation is easy to analyze. The Disco worker submits (key, value) pairs to the external Map implementation via STDIN and retrieves the results via STDOUT. The behavior of the Disco worker is therefore easy to simulate.

The data passed to the external Map are in the format presented in Table 12.2. After the external Map is executed, the Disco worker first sends an zero terminated string expressing a number, the `nparams` as seen in the first line of the table. The number specifies the length of the parameter set passed by the Disco worker to the external Map process. This operation is done exactly once during the lifetime of the external Map process. The implementation of the external Map in this text does not use the parameters, yet to conform with the Disco behavior, the parameters must be supplied. It is legal to pass a zero terminated string '0'.

Next, for each (key, value) pair, the Disco worker emits the (key, value) pair to the STDIN. The format in which the (key, value) pair is emitted is presented

```
struct stdin_data {
    /* # of params from Disco itself (null-terminated string) */
    char        number_of_params[];
    /* List of params from Disco itself (null-terminated string) */
    char        params[];

    /* Length of key, LSB first (without trailing zero) */
    uint32_t    klen;
    /* The key (null-terminated string) */
    char        key[];

    /* Length of value, LSB first (without trailing zero) */
    uint32_t    vlen;
    /* The value (null-terminated string) */
    char        value[];
};
```

Figure 12.2: Disco external Map process STDIN data format

by the last two lines of the table. First is the 4-byte length of the key followed by the key, then 4-byte length of the payload followed by the payload.

Such data can be easily generated using shell. From the FPGA perspective, the contents of the `data` is completely irrelevant, therefore these data can be zeroes. The key is the letter 'k', since this is defined in the benchmark and the length of the key is thus one byte. The length of the payload is 1MiB, since the payload is comprised of 2^{18} 4-byte entries.

```
npairs=128

echo -ne "0\0" > stdin_data.bin
echo -ne "\x01\x00\x00\x00k\0\x00\x10\x00\x00" | \
        cat - /dev/zero | dd bs=1048585 count=1 > pair.bin

for i in `seq 1 ${npairs}` ; do
        cat pair.bin >> stdin_data.bin
done
```

Figure 12.3: Script to generate testing data for external Map process

The tested input data for the external Map are generated using the commands in Figure 12.3. The result is stored in `stdin_data.bin` file and contains `npairs` (key, value) pairs.

To evaluate if the performance problem is in the implementation of the external Map itself, the duration of execution of the external Map with the generated input data is tested using the command in Figure 12.4.

Section 7.2.2 on page 29 pointed out that it should take around $0.005s$ to compute 2^{18} MD5 blocks using the FPGA. The sample data contain 128 blocks of (key, value) pairs, where each block contains exactly 2^{18} of values, each of

```
$ time ./md5 map < stdin_data.bin >/dev/null
[...]
real    1m0.392s
user    0m59.600s
sys     0m0.648s
```

Figure 12.4: Evaluation of duration of the external Map implementation

which is used for a single MD5 block. The expected duration of the computation is therefore $(128 * 0.005)s = 0.64s$, but the computation takes over 1 minute instead[1].

In order to determine where exactly does the performance disappear, the program must now be profiled. The profiler of choice is on a GNU/Linux system is the `gprof(1)`. The target program must first be re-compiled to enable debugging symbols and profiling information. This is done by passing the `-pg -g` options to the GCC. It is expected that the recompiled binary will be slower, since the `-pg` option adds extra code into the binary to generate and store the profiling information.

```
$ gprof -p md5 gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls    s/call   s/call  name
 55.21     17.11    17.11 67108864     0.00     0.00  write_entry
 18.52     22.85     5.74      128     0.04     0.07  md5_set_blocks
 11.55     26.43     3.58 33554432     0.00     0.00  md5_set_block
 11.20     29.90     3.47      128     0.03     0.16  md5_write_output
  3.45     30.97     1.07      256     0.00     0.00  fpga_dma_poll
  0.06     30.99     0.02  3206864     0.00     0.00  readl
  0.00     30.99     0.00     1024     0.00     0.00  writel
  0.00     30.99     0.00      260     0.00     0.00  msg
  0.00     30.99     0.00      256     0.00     0.00  dxmalloc
  0.00     30.99     0.00      256     0.00     0.00  fpga_dump_dma_state
  0.00     30.99     0.00      256     0.00     0.00  read_pentry
  0.00     30.99     0.00      129     0.00     0.07  md5_read_input
  0.00     30.99     0.00      129     0.00     0.00  read_kv
  0.00     30.99     0.00      128     0.00     0.00  fpga_dma_setup
  0.00     30.99     0.00      128     0.00     0.00  fpga_dma_start
  0.00     30.99     0.00      128     0.00     0.01  fpga_loop
  0.00     30.99     0.00      128     0.00     0.00  write_num_prefix
  0.00     30.99     0.00        1     0.00    30.99  process
  0.00     30.99     0.00        1     0.00     0.00  read_parameters
```

Figure 12.5: gprof statistics of the external Map process

[1]The data provided by the `time(1)` command are not accurate, see the section ACCURACY of the `time(1)` manual page for details

The Map binary is now executed with the same input data as above again. Once the Map binary finishes it's execution, the profiling code in the binary generates a profiling report in a file `gmon.out`. The analysis of the report using `gprof` is presented in Figure 12.5. By comparing the `gprof` output with the source code of the external Map function implementation, it is now possible to identify the bottleneck.

The first entry shows that the vast majority of execution time is wasted by emitting data to STDOUT. By analyzing the `write_entry()` function, which is part of the Disco external C interface, it becomes clear that the function only does two calls to the `fwrite()` function.

It is surprising to see how many times this function is called and how very small the data that are written using this function are. This particular fact itself pointed out a grave issue in the external Map() function implementation during it's development and is a reason why the `md5_write_output()` function uses two calls to `write_entry()` to emit the (key, value) pair instead of using single `write_kv()` calls.

The `write_kv()` implementation does also internally call `write_entry()` twice, but in addition to that, it calls `fflush(stdout)` at the end, which circumvents the positive impact of the caching in `fwrite()`.

The implementation of the external Map function here does the flush all the way at the end of `md5_write_output()` function, which is legal, since the Disco python interface to the external worker is informed about the amount of (key, value) pairs that will be submitted to it from the external Map worker using the `write_num_prefix()` function. The adjusted external Map implementation is on the attached CD in the `fpga/src/sw/disco/md5.ext` directory and the patch demonstrating the change to avoid the excessive STDOUT flushing is in `fpga/src/sw/disco/0001-MD5-Avoid-unneeded-stdout-flushing.patch` file.

The subsequent three entries in the profiler output, the `md5_set_blocks`, `md5_set_block` and `md5_write_output()` trigger a lot of random accesses into the uncached memory area used to exchange data and results between the CPU and the FPGA. Accessing the uncached memory has a huge performance penalty.

```
[9]    3.5     0.00     1.09     128          fpga_loop [9]
               1.07     0.02     256/256          fpga_dma_poll [8]
               0.00     0.00     256/256          fpga_dump_dma_state [14]
               0.00     0.00     128/128          fpga_dma_setup [17]
               0.00     0.00     128/128          fpga_dma_start [18]
```

Figure 12.6: gprof statistics of the external Map process, FPGA interface detail

On a final note, Figure 12.6 is an output from the profiler which shows call graph of the `fpga_loop()` function. The function is responsible for starting the transformation of the whole block of data on the FPGA and waiting for the FPGA to finish by polling it's control registers. It is clearly seen that the function is called once for every entry in the input data and that the sum of time spent executing the function is 1.09 seconds. The duration of a single execution of the function is therefore $\frac{1.09s}{128} = 0.0085s$. This value is indeed much closer to the maximum theoretical performance of the FPGA core, which is 0.005 seconds according to Section 7.2.2 on page 29.

Conclusion drawn from this section is that the performance of the Map() function has been pushed to it's limit. The main bottleneck is in emitting data to the STDOUT and in moving data to and from the uncached RAM area.

12.2 Framework overhead

It is possible to enable status reporting to a console from which the Disco job was executed. If this feature is enabled, the Disco will emit job progress report to the console every second. This feature is enabled for the reference benchmark already, since it is used to gather information about number and type of running tasks in the cluster. This report can be used to gain the high level overview of the behavior of the Disco job.

```
... 02:25:34 mcvevk MSG: [map:0] input: ret=0 vals=262144 nchunk=0
... 02:25:34 mcvevk MSG: [map:0] output: ret=0 vals=262144 nchunk=0
... 02:26:18 mcvevk MSG: [map:0] input: ret=0 vals=262144 nchunk=1
... 02:26:18 mcvevk MSG: [map:0] output: ret=0 vals=262144 nchunk=1
... 02:27:05 mcvevk MSG: [map:0] input: ret=0 vals=262144 nchunk=2
... 02:27:06 mcvevk MSG: [map:0] output: ret=0 vals=262144 nchunk=2
... 02:27:52 mcvevk MSG: [map:0] input: ret=0 vals=262144 nchunk=3
... 02:27:52 mcvevk MSG: [map:0] output: ret=0 vals=262144 nchunk=3
... 02:28:38 mcvevk MSG: [map:0] Done: 4 entries mapped
... 02:28:40 mcvevk MSG: [map:0] Results sent to master
... 02:28:40 mcvevk DONE: [map:0] Task finished in 0:03:09.338
```

Figure 12.7: Initial Disco log from the FPGA-assisted Map-Reduce job

The external Map worker code is augmented to emit messages into this log[2]. The external Map worker emits two messages per each (key, value) pair, one message at the beginning of the processing and another at the end. An initial observation of the relevant part of the Disco log[3] is in Figure 12.7.

The log clearly shows that it takes approximately 45 seconds to complete one (key, value) pair transformation, which renders the combined ARM-FPGA system unusable for the Map-Reduce use. The previous section pointed out that for the external Map implementation alone, it takes roughly $0.0085s$ to complete 1 (key, value) pair transformation. It is therefore of interest to learn where do these additional 45 seconds come from.

```
%CPU %MEM    VSZ   RSS STAT START TIME COMMAND
93.8  2.3  32168 23796  RNs 02:25 0:30 \_ python [...]/classic/worker.py
 1.4  0.1 136988  1900  SN  02:25 0:00     \_ ext.map/op map
```

Figure 12.8: Process listing the FPGA-assisted Map-Reduce job on the Node

Portion of the process tree on the ARM-FPGA machine showing both the Disco python worker and the external Map worker is presented in Figure 12.8.

[2] `fpga/src/sw/disco/md5.ext/md5.c`, see `msg()` calls in the `process()` function

[3] Full log is on the attached CD in `disco/fpga/profile/mr-0.log`

It is clearly seen that the Disco worker is utilizing the CPU, while the external Map worker, represented by the `ext.map/op` process is almost idle. This raises suspicion that the overhead is coming from the Disco worker or the Python itself.

The Disco framework supports regular Python cProfiler to profile the behavior of the tasks. Profiling is enabled by passing the `profile=True` parameter into the `job.run()` method.

Since the problem is located in the Map() function of the Disco job, it is beneficial to eliminate the overhead of the Reduce() function and of printing out the results from the computation. The Reduce() function is eliminated by passing `reduce=None` argument to the `job.run()` method, which makes the job terminate after the Map() function finished. The printing of the results is eliminated in the Disco Job's main function.

Please note that for convenience purpose, the reference benchmark implementation has a special mode of operation, in which all these profiling features are enabled. The benchmark is started in this profiling mode by starting the benchmark with the `"fpgaprof"` parameter.

When using the `"fpgaprof"` mode, the results from the profiler are dumped into a file called `/work/disco/mr.stats` after the Job completed it's execution. The file can be loaded from a regular Python shell and analyzed further at a later date.

Since it was observed in Subsection 9.1.3 on Page 48 that a single run of the benchmark takes roughly 8 hours on the ARM-FPGA system, it is beneficial to introduce a smaller set of input data for the purpose of the profiling. Doing so is legal, since the profiling is trying to determine why does one loop of the FPGA operation take 45 seconds instead of a fragment of a second. The smaller set of data used to do the profiling in this section has exactly 4 (key, value) pairs, with 1 value containing 2^{18} inputs.

```
$ python
...
>>> import pstats
>>> p=pstats.Stats("mr.stats")
>>> p.sort_stats('time').reverse_order().print_stats()
...
 ncalls tottime percall cumtime  percall filename:lineno(function)
...
1048577  10.466   0.000  14.075    0.000 worker.py:414(opener)
1048576  10.643   0.000  46.879    0.000 fileutils.py:82(append)
1048576  11.584   0.000  18.326    0.000 fileutils.py:110(hunk_write)
1048576  11.976   0.000  11.976    0.000 {cPickle.dumps}
1048576  16.574   0.000  34.543    0.000 worker.py:338(output)
4194324  24.378   0.000  24.378    0.000 {method 'read' of 'file' objects}
1048576  25.882   0.000  55.160    0.000 external.py:370(unpack_kv)
      1  28.187  28.187 185.803  185.803 worker.py:331(map)
```

Figure 12.9: Initial cProfiler profile of the FPGA-assisted Mapper

An example of loading the output from the profiler using Python shell, displaying the results and the results are in Figure 12.9. The irrelevant portions of

the results have been redacted out[4].

It is clearly seen that the `map` method, which is defined in `lib/disco/worker/classic/worker.py` introduces the most overhead. The `map` method must thus be further scrutinized.

The body of the generic `map` method iterates over each (key, value) pair that is received from the user-supplied `map` method. In this case, the supplied method calls the external interface, which in turn communicates with the external Map implementation. The `map` method applies partitioning function and combining function on each (key, value) pair in this process. Finally, the (key, value) pair is serialized and written to the partition determined by the partitioning function.

Please note that the approach taken further, which involves modifying the Disco framework itself is not advisable. The modification of the Disco framework is done purely to determine where the overhead is coming from.

Since the format of the data on which the `map` method operates is well defined and the Map-Reduce Job is also well defined, it is possible to bypass most of the logic of the generic `map` method. A patch for the `map` method is on the attached CD in the `disco/fpga/patches/0001-Disco-Simplify-map-method.patch` file.

The patch bypasses the logic where the generic `map` method iterates over each (key, value) pair that is received from the external interface and applies the partitioning function. Instead, the entire bulk of (key, value) pairs is immediately serialized and written to the partition.

The partitioning function had to be replaced in this case. Originally, the partitioning function was applied to each (key, value) pair separately, but the modified `map` method operates on the whole set of (key, value) pairs. The partitioning function was replaced by a simple up-counter, which wraps at the maximum number of partitions. This thus retains the ability of distributing the data evenly across partitions.

```
 ncalls tottime percall cumtime percall filename:lineno(function)
...
1048580   5.062   0.000  56.697   0.000 external.py:396(communicate)
      1   6.986   6.986  82.233  82.233 worker.py:331(map)
      4   7.328   1.832   7.328   1.832 {zlib.compress}
1048576   9.864   0.000   9.864   0.000 {cPickle.dumps}
4194324  21.979   0.000  21.979   0.000 {method 'read' of 'file' objects}
1048576  23.289   0.000  49.774   0.000 external.py:370(unpack_kv)
```

Figure 12.10: cProfiler profile of the FPGA-assisted Mapper with Map function fix

The resulting profile[5] after applying the fix to the `map` method is presented in Figure 12.10. The overhead of the `map` method dropped from 28 seconds to 7 seconds, which is remarkable. The Disco log[6] shows that the duration of processing 1 (key, value) pair dropped from 45 to 20 seconds as well. This clearly shows that there is a problem with using Python on ARM.

[4]Full cPython profile is on the attached CD in `disco/fpga/profile/mr-0-orig.stats`

[5]Full cPython profile is on the attached CD in `disco/fpga/profile/mr-1-mapfix.stats`

[6]Full log is on the attached CD in `disco/fpga/profile/mr-1.log`

The overhead has shifted from the `map` method to the `unpack_kv` method defined in `lib/disco/worker/classic/external.py`. Furthermore, the line above `unpack_kv` is also suspicious. Remember that there are 4 (key, value) pairs in the data set used for the profiling and there is 2^{18} inputs in each value. This produces a total of 1048576 (key, value) pairs emitted the external Map() implementation.

By analyzing the `unpack_kv` method, it is seen that the method reads STDOUT of the external Map() implementation four times. Since the format of data emitted by the external Map() implementation is fixed and known, it is possible to reduce this to 1 read only.

```
 ncalls tottime percall cumtime percall filename:lineno(function)
...
1048580   5.007   0.000  20.054   0.000 external.py:398(communicate)
1048576   6.566   0.000  13.194   0.000 external.py:370(unpack_kv)
      1   6.582   6.582  45.438  45.438 worker.py:331(map)
1048596   6.642   0.000   6.642   0.000 {method 'read' of 'file' objects}
      4   8.242   2.060   8.242   2.060 {zlib.compress}
1048576   9.190   0.000   9.190   0.000 {cPickle.dumps}
```

Figure 12.11: cProfiler profile of the FPGA-assisted Mapper with unpack_kv fix

A patch to reduce the number of STDOUT reads is on the attached CD in the `disco/fpga/patches/0002-Disco-Simplify-external-unpack_kv.patch` file and the result from the profiler[7] with this patch is presented in Figure 12.11. The time spent by executing the `unpack_kv` method dropped over 3 times. The amount of STDOUT reads was reduced by a factor of four and in turn reduced time spent by reading STDOUT of the external Map process 3 times. The Disco log[8] shows that the duration of processing 1 (key, value) pair dropped from 20 to 11 seconds too.

This shows that interfacing the external Map() process via STDOUT is suspicious. One last attempt to reduce the number of reads of STDOUT was conducted by completely bypassing the `unpack_kv` method. Instead of reading the (key, value) pairs emitted by the external Map() implementation separately, all of them are read in bulk and this long block of data is then divided and emitted at once. Again, a patch is on the attached CD in the `disco/fpga/patches/0003-Disco-Bypass-unpack_kv.patch` file.

The final result from the profiler[9] is presented in Figure 12.12. The Disco log[10] shows that the duration of processing 1 (key, value) pair dropped further from 11 to 7 seconds. Compared to the initial observations, where the processing took 45 seconds, the improvement is over 600 percent.

The profiler output clearly shows that the remaining overhead is hidden mostly in the Python and it's components, as exclaimed by the dominance of `zlib`, `cPickle` and invocations of the methods on the `file` object. The remaining overhead in the `map` method and `communicate` method is also introduced by heavy Python functions used within.

[7]Full cPython profile is on the attached CD in`disco/fpga/profile/mr-2-kvfix.stats`
[8]Full log is on the attached CD in `disco/fpga/profile/mr-2.log`
[9]Full cPython profile is on the attached CD in`disco/fpga/profile/mr-3-kvbypass.stats`
[10]Full log is on the attached CD in `disco/fpga/profile/mr-3.log`

```
 ncalls tottime percall cumtime percall filename:lineno(function)
...
     24   1.744   0.073   1.744   0.073 {method 'read' of 'file' objects}
     48   1.811   0.038   1.811   0.038 {method 'write' of 'file' objects}
1048580   3.965   0.000   7.543   0.000 external.py:398(communicate)
      1   5.996   5.996  30.084  30.084 worker.py:331(map)
      4   7.337   1.834   7.337   1.834 {zlib.compress}
1048576   7.878   0.000   7.878   0.000 {cPickle.dumps}
```

Figure 12.12: cProfiler profile of the FPGA-assisted Mapper with unpack_kv bypass

12.3 Re-evaluating the performance

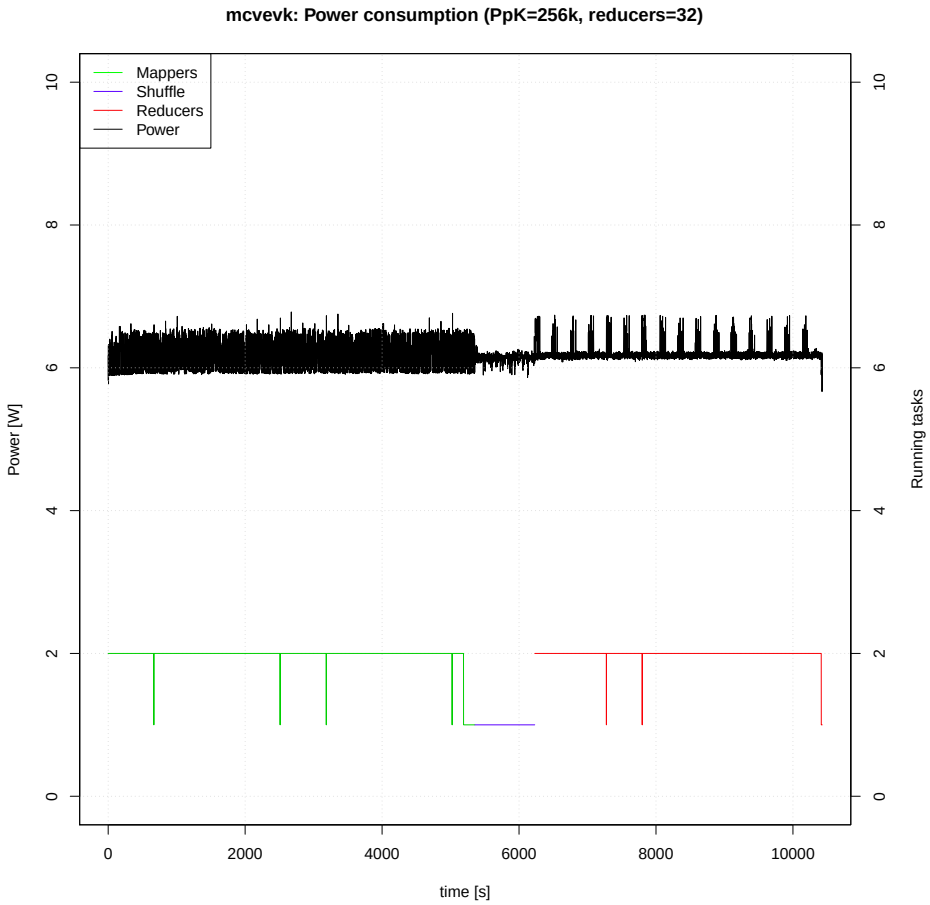


Figure 12.13: Consumption: 1x ARM-FPGA, 2^{18} PpK, 32 Mappers, 32 Reducers

The performance of all the testbed systems was re-evaluated for a 2^{18} PpK, 32 Reducer configuration on a single node. The re-evaluation was necessary, since the changes to the Disco framework are not isolated to the Disco external interface used to communicate with the external Map process, but also affect the core Disco code. Without re-running the tests, it would not be possible to have a common framework baseline and the results would not be comparable.

The results of the automated testing are presented in Table 12.1 and Table 12.2. A graphical representation of a single ARM-FPGA node test is presented in Figure 12.13. All of the measurement artifacts are on the attached CD in the `measurements/data/mcvevk-patched`, `measurements/data/novena-patched` and `measurements/data/x86-patched` directories for ARM-FPGA, ARM Novena and reference x86 system respectively.

2^{18} PpK				
Hardware	$t_{map}[s]$	$t_{shuffle}[s]$	$t_{reduce}[s]$	$t_{total}[s]$
ARM-FPGA	5351.4 ± 19.05	869.6 ± 65.37	4197.6 ± 3.98	10418.6 ± 67.39
ARM Novena	1808.4 ± 14.77	856.6 ± 37.57	1561.6 ± 4.34	4226.6 ± 50.51
x86 system	115.4 ± 1.14	18.4 ± 0.55	113 ± 1.58	246.8 ± 2.39

Table 12.1: Timing for patched Disco: 1x system, 32 Mappers, 32 Reducers

Hardware	$TpW_{2^{18}}$
ARM-FPGA	4180.1 ± 17.72
ARM Novena	6625.02 ± 42.75
x86 system	4104.92 ± 19.5

Table 12.2: TpW for patched Disco: 1x system, 32 Mappers, 32 Reducers

It is immediately clear by comparing the original evaluation of the ARM-FPGA system in Subsection 9.1.3 on page 48 and the results in the tables in this subsection that even trivial changes to the Disco Map-Reduce framework have tremendous impact on the TpW ratio of the ARM-FPGA system and the runtime of the Map stage, in which the FPGA accelerates the processing. The TpW ratio increased from 1306.98 to 4180.1, which is an improvement of more than factor three. The duration of the Map stage dropped from 27117 seconds to 5351.4 seconds, which is over five times shorter. The total runtime of the job dropped from 32317 seconds to 10418.6 seconds because of the improvements.

The impact of the changes on the ARM Novena and the reference x86 system is also significant when compared to the measurements without the changes in place in Chapter 10 on page 52. Using the modified Map function and related changes decrease the duration of the Map stage on ARM Novena by a factor of 2.8, in total 1808.4 seconds compared to the 5215.8 without. A similar figure is observed on the reference x86 system, where the Map stage is 2.7 times faster, 115.4 seconds compared to 319.4 seconds. This shows that the Disco framework itself has a tremendous overhead, resulting in a severe loss of performance.

Finally it should also be clear that the absolute improvement in runtime for the ARM-FPGA system is by far the largest. This was however expected as the analysis shows that the original test case suffered from two major impediments. Fixing the generic problem in the Map phase also improves the runtime for the other tests, but fixing the severe external mapper bottleneck of course only profits the ARM-FPGA case.

It is even noticable that such growth puts the TpW of the ARM-FPGA system on-par with the reference x86 system. This implies that the ARM-FPGA system has a large performance potential, which is currently diminished by the overhead of the Disco framework itself.

Conclusion

The foremost goal of this work was to find a more power efficient solution for a Map-Reduce cluster. This goal then further boiled down to three smaller steps. The first step was to integrate an ARM and an ARM-FPGA system into a Map-Reduce framework, the next step was to identify problems with using ARM and ARM-FPGA systems in a Map-Reduce context and the final step was to evaluate whether it is viable to use either of such systems in the Map-Reduce context.

Both of the ARM and ARM-FPGA systems were integrated into the Disco Map-Reduce implementation. This integration of an ARM system is absolutely straightforward and there is no problem at all even though the ARM system has a non-standard architecture. This implies that a mixed-architecture cluster is possible, at least in the realm of the Disco Map-Reduce implementation.

The integration of an ARM-FPGA system resulted in a generic implementation of a tool called FPGAd that allows a user process to accelerate computation using an FPGA without having to worry about other processes that might require the same. FPGAd arbitrates access to the FPGA by user processes at any given time. While the tool was written with the intention of evaluating the performance of the ARM-FPGA system in the Disco Map-Reduce context, the tool is generic enough to be used elsewhere as well.

Both of the ARM and ARM-FPGA systems were evaluated in the context of the Disco Map-Reduce framework for a defined MD5 benchmark. For this particular benchmark, the ARM Novena system showed a much higher power efficiency then the reference x86 system, both in a single-node setup as shown in Chapter 9 and Chapter 10 and in a cluster setting as shown in subsequent Chapter 11. In all cases, the ARM Novena showed over 1.5 times higher TpW ratio than the reference x86 system, yet the ARM Novena also took over 15 times longer to finish the computation in a single-system setting.

#Partitions	x86 $TpW_{2^{18}}$	ARM $TpW_{2^{18}}$
8	2233.96 ± 8.73	3783.25 ± 30.41
16	2220.78 ± 24.81	3600.06 ± 24.36
24	2229.98 ± 10.01	3691.47 ± 48.21
32	2263.85 ± 9.95	3721.68 ± 31.32

TpW: 1x ref. x86 vs 1x ARM Novena, 32 Mappers, 32 Reducers

The ARM-FPGA system initially showed a very disappointing results, where the TpW ratio was 1306.98 and the computation took 32317 seconds, which is detailed in Subsection 9.1.3. The performance problem was identified and dissected in Chapter 12. The Disco Map-Reduce framework was adjusted to bypass the problem and a subsequent re-evaluation of all systems was conducted.

With such adjustments in place, the duration of the *Map* stage decreased in case of both ARM Novena and the reference x86 system approximatelly 2.7 times, but in case of the ARM-FPGA system, the duration decreased over 5 times. This immediately implies that a comparable adjustment, which decreases the framework overhead and thus allows better saturation of the computational units, yields far better results on the ARM-FPGA system.

Hardware	$t_{map}[s]$ (stock Disco)	$t_{map}[s]$ (patched Disco)
ARM-FPGA	27117	5351.4 ± 19.05
ARM Novena	5215.8 ± 100.61	1808.4 ± 14.77
x86 system	319.4 ± 3.44	115.4 ± 1.14
Hardware	$TpW_{2^{18}}$ (stock Disco)	$TpW_{2^{18}}$ (patched Disco)
ARM-FPGA	1306.98	4180.1 ± 17.72
ARM Novena	3721.68 ± 31.32	6625.02 ± 42.75
x86 system	2263.85 ± 9.95	4104.92 ± 19.5

Timing and TpW comparison: 1x system, 32 Mappers, 32 Reducers

The ARM-FPGA system also shows over 3 times improvement in the TpW ratio. Such improvement puts the ARM-FPGA system and the reference x86 system are on-par in terms of TpW ratio if the Disco framework is patched to bypass it's overhead.

The substantial overhead of the Disco Map-Reduce framework, respectively the Python language makes this framework a bad choice for using Map-Reduce on power-efficient ARM systems. An ARM system would certainly benefit from using a Map-Reduce framework with less overhead in its implementation.

Summarily this work has shown that ARM and especially ARM-FPGA systems clearly have the potential to yield better TpW values for Map-Reduce clusters. Achieving this goal is however not a trivial operation that can be summarized in an easy recipe. There are a lot of parameters entering the final TpW value and all of those have to be selected carefully. Especially for the ARM-FPGA system some more groundwork will have to be done to incorporate customizable hardware into the Map-Reduce framework. Although the FPGA configuration itself can be seen as "yet another program", the details on how to use it involve the hardware / software interface and thus need work on the operating system level itself. In this work, a Linux kernel module and a suitable user space deamon was developed as a userspace API for this task. The concepts used therein are generic enough to be applied to different settings but that needs to be evaluated on at least a different CPU + FPGA platform.

Future work

The observations made in this text only present the first step. By taking this first step, it is now possible to determine which directions can be taken further and which of them provide promising results.

This work can be immediatelly followed by evaluating different Map-Reduce frameworks and their overhead on ARM. It is also possible to research the overhead introduced by the Disco framework further, it would certainly be interesting to learn whether using a Python JIT[11], like PyPy[12] or Pyston[13], can improve the performance of the Disco framework itself.

Section 12.1 shows that even though a lot of optimization happened, there is an implicit overhead when moving data between the ARM and the FPGA because of the non-cacheable RAM. Modern FPGAs support streaming mode, where the data pass through the FPGA. These FPGAs also support packet transfer and it is even possible to tunnel ethernet into the FPGA itself. It would therefore be of great interest to research if it is possible to put the ARM core in the cluster node into a control-only function and somehow tunnel the packets from DDFS directly through the FPGA, which would do the transformation on the data in the packets and report the result directly back to the DDFS.

A typical 19-inch 1U rack blade is 44.45 mm high, 482.6 mm wide and 736.6 mm deep. An ARM based computer requires as small space as 95mm wide and 95mm deep. Such a 1U rack blade would thus be able to host about 35 of such ARM systems, compared to 1 or 2 x86 systems. It is also possible to have all 35 such ARM systems in a local network implemented on the main board in the blade. It would be interesting to physically build such a blade and evaluate it's performance.

Finally, it would be interesting to evaluate the upcoming 64-bit ARMv8 architecture, which is completely different from the 32-bit ARMv7 used in this text.

[11] Just-in-time compiler

[12] `http://pypy.org/`

[13] `https://github.com/dropbox/pyston`

Bibliography

[1] J. Dean and S. Ghemawat,
MapReduce: Simplified Data Processing on Large Clusters.
OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[2] U. Drepper
What Every Programmer Should Know About Memory
`http://www.akkadia.org/drepper/cpumemory.pdf` (retrieved on 2014-11-30)

[3] M. Grossman, M. Breternitz, V. Sarkar
HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL
IPDPSW '13 Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, Pages 1918-1927

[4] C. He, P. Du
CUDA Performance Study on Hadoop MapReduce Clusters
`http://www.slideshare.net/airbots/cuda-29330283` (retrieved on 2014-11-30)

[5] Z. Krpić, G. Horvat, D. Žagar and G. Martinović
Towards an energy efficient SoC computing cluster
Proceedings of the 37th International Convention MIPRO, 2014, pp. 178-182

[6] Z. Ou; B. Pang; Y. Deng; N., J.K.; Ylä-Jääski, A.; P. Hui
Energy- and Cost-Efficiency Analysis of ARM-Based Clusters
Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on , vol., no., pp.115,123, 13-16 May 2012

[7] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, A. Ramirez
Tibidabo: Making the case for an ARM-based HPC system,
Future Generation Computer Systems, Available online 7 August 2013.

[8] R. Rivest
The MD5 Message-Digest Algorithm
IETF RFC 1321, April 1992
`http://www.ietf.org/rfc/rfc1321.txt` (retrieved on 2014-11-30)

[9] Y. Wang
Hadoop MapReduce Performance Study on ARM cluster
`http://www.slideshare.net/airbots/hadoop-mapreduce-performance-study-on-arm-cluster` (retrieved on 2014-11-30)

[10] H. Yan
The first Spark/Hadoop ARM cluster runs atop Cubieboards
`http://www.cubietruck.com/blogs/news/13689917-the-first-spark-hadoop-arm-cluster-runs-atop-cubieboards` (retrieved on 2014-11-30)

[11] D. Yin; G. Li; K. Huang
Scalable MapReduce Framework on FPGA Accelerated Commodity Hardware
12th International Conference, NEW2AN 2012, and 5th Conference, ruSMART 2012, St. Petersburg, Russia, August 27-29, 2012. Proceedings, Book II, pp. 280-294

[12] L. Zhongduo and P. Chow
ZCluster: A Zynq-based Hadoop cluster
Field-Programmable Technology (FPT), 2013 International Conference on, pp. 450-453

[13] *Apache Hadoop*
`http://hadoop.apache.org/` (retrieved on 2014-11-30)

[14] *Bash-Reduce*
`https://github.com/erikfrey/bashreduce` (retrieved on 2014-11-30)

[15] *Bash-Reduce Documentation*
`https://raw.githubusercontent.com/erikfrey/bashreduce/master/README.textile` (retrieved on 2014-11-30)

[16] *DENX MCVEVK Technical Documentation, 6.2.3 Clocking*
`http://www.denx-cs.de/doku/?q=mcvclocking` (retrieved on 2014-11-30)

[17] *Disco Map-Reduce*
`http://discoproject.org/` (retrieved on 2014-11-30)

[18] *Frequently Asked Questions about the GNU C Library*
Even statically linked programs need some shared libraries which is not acceptable for me. What can I do?
`https://sourceware.org/glibc/wiki/FAQ#Even_statically_linked_programs_need_some_shared_libraries_which_is_not_acceptable_for_me.__What_can_I_do.3F` (retrieved on 2014-11-30)

[19] J. Leitch
MD5 Pipelined
`http://opencores.org/project,md5_pipelined` (retrieved on 2014-11-30)

[20] *Mars Map-Reduce*
`http://www.cse.ust.hk/gpuqp/Mars.html` (retrieved on 2014-11-30)

[21] *Modular SGDMA - Altera Wiki*
`http://www.alterawiki.com/wiki/Modular_SGDMA` (retrieved on 2014-11-30)

[22] *PoweredBy - Hadoop Wiki*
`http://wiki.apache.org/hadoop/PoweredBy` (retrieved on 2014-11-30)

[23] *Python Global Interpreter Lock*
`https://wiki.python.org/moin/GlobalInterpreterLock` (retrieved on 2014-11-30)

[24] *Twister Iterative Map-Reduce*
`http://www.iterativemapreduce.org/` (retrieved on 2014-11-30)

List of Figures

List of Tables

List of Abbreviations

CMA Contiguous Memory Allocator
DDFS Disco Distributed File System
DMA Direct Memory Access
FPGA Field-Programmable Gate Array
GIL Global Interpreter Lock
HPS Hard Processor System , CPU component of the SoCFPGA chip
IV Initialization Vector (for MD5 hash)
LSB Least Significant Byte
MD5 MD5 Message-Digest Algorithm
MSB Most Significant Byte
PpK Plain text per Key (Section 8.4 on page 43)
PpW Performance-per-Watt
SoM System-on-Module
TpW Transformations-per-Watt (Chapter 6 on page 23)
UPS Uninterruptible Power Supply
USB Universal Serial Bus

Contents of the CD

The attached CD contains README file describing the structure of the top-level directory as well as a README file in each subdirectory of the top-level directory to make the navigation easier. Here is the list of important directories:

disco/	
disco/fpga	Patches and profiling artifacts used in Chapter 12 for the ARM-FPGA system.
disco/job	The benchmark implementation in the Disco Map-Reduce framework
fpga/	
fpga/bin	Precompiled binaries for the FPGA multiplexing daemon (FPGAd), the external Map process which uses the FPGA and the FPGA bitstream.
fpga/lib	Precompiled client library for the FPGA multiplexing daemon (FPGAd).
fpga/src/hw	FPGA project with the integrated MD5 IP block
fpga/src/sw	Source code for the FPGA multiplexing daemon (FPGAd), external Map process which uses the FPGAd to accelerate the MD5 computation and a Linux kernel patch to allow FPGAd to access to the FPGA.
measurements/	
measurements/bin/	Scripts and precompiled binaries used to conduct the measurements on all systems in this text.
measurements/data/	Raw measurement data, intermediate pre-processed data, plots for all samples and final statistics.
measurements/src/	Source code for the UTC tool, which is used to dump samples from the scope.